Detecting the Misuse of Secrets:

Foundations, Protocols, and Verification

Kevin Milner

Wolfson College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Hilary 2018

Acknowledgements

First and foremost, I would like to thank my supervisor, Cas Cremers, for his advice, expertise, and patience. His guidance is the reason I have had the chance to research such fascinating topics, and my work is better for his repeated encouragement to 'do good science'.

I must also thank the rest of the information security research group at Oxford: Katriel Cohn-Gordon, Martin Dehnel-Wild, Luke Garratt, Dennis Jackson, and Nick Moore, as well as our 'honorary members' Chad Heitzenrater and Yang Liu. You have all been an important part of my time in Oxford, and I am happy I've had the chance to work (and not work) with all of you.

I want to recognize the many people involved with the Centre for Doctoral Training in Cyber Security at Oxford: the director, Andrew Martin, without whom the CDT would not be nearly the same—or 'be' at all—and the administrators, Maureen York and David Hobbes, who were always helpful and put up with more than they should have to (I still have a copy of that key). And of course, I must thank the entire CDT 2013 cohort, who have made a huge difference to my time in Oxford; I expect you will all go on to do great things.

I would like to thank the many friends and family, near and far, who have supported me directly or indirectly over my time in Oxford. I cannot hope to contain all your names here, but I am very thankful to know so many wonderful people in so many places.

Finally, I would like to thank my partner Teri Drummond, who has been endlessly supportive throughout my research despite the time we have had to spend apart.

Abstract

Secrets are the basis of most protocol security, enabling authentication and secrecy over untrusted channels even in the presence of active adversaries. The compromise and misuse of secrets can therefore undermine the properties that people and systems rely on for their security. In this thesis, we develop foundations and constructions for security protocols that can automatically detect, without false positives, if a secret such as a key or password has been misused. These constructions allow protocol participants to automatically trigger an appropriate response and minimize the effects of compromise. Our threat model includes malicious agents, (temporarily or permanently) compromised agents, and clones.

Unlike existing approaches to detection, for which designs are interwoven with domain-specific considerations and which usually do not enable fully automatic response (i.e., they need human assessment), our approach makes it clear where automatic action is possible. Our results unify, justify the design of, and suggest improvements for existing domain-specific solutions. For example, we propose an improvement to Cloudflare's Keyless SSL protocol that enables key misuse detection, and a modified ISO-IEC 9798 protocol for use in high-security smart-card applications such as the Common Access Card.

Our results show that protocols which detect misuse must be stateful. Although stateful protocols are becoming more common, formally analyzing them remains a challenge. We develop and implement several improvements to the Tamarin prover to alleviate this difficulty, and use them to formally verify our proposed detection protocols.

We demonstrate the wide applicability of our improvements to Tamarin by analyzing several real-world protocols. Notably, we perform the first full formal analysis of both WireGuard, and the DNP3 Secure Authentication protocol—a complex multi-stage stateful protocol used to authenticate critical monitor and control functionality in SCADA systems.

Contents

1 Introduction **1**
1.1 Approach . . . 3
1.2 Contributions . . . 6
1.2.1 Originality and other contributions . . . 6
1.3 Overview . . . 7

2 Background and related work **9**
2.1 Security protocols . . . 9
2.1.1 The Dolev-Yao model . . . 10
2.1.2 Compromising adversaries . . . 11
2.1.3 Stateful protocols . . . 12
2.2 Restoring security after compromise . . . 13
2.2.1 Key-evolving cryptosystems . . . 13
2.2.2 TPM authentication protocols . . . 14
2.2.3 Key ratcheting and post-compromise security . . . 14
2.3 Detection of misuse . . . 15
2.3.1 Clone detection methods . . . 15
2.3.2 Causality-based fault detection . . . 17
2.3.3 Transparency overlays . . . 18
2.3.4 Accountability and verifiability . . . 19

3 Foundations of detection **21**
3.1 Motivating examples . . . 21
3.2 Notation and framework . . . 26
3.2.1 Notation . . . 26
3.2.2 Reasoning about protocols . . . 26
3.2.3 Reasoning about agents . . . 30

3.3 Observation of misuse 33
3.3.1 Categorizing observable misuse 35
3.4 Design implications and applications 41
3.4.1 Main detection mechanisms 41
3.4.2 Design principles 42

4 Applications of detection theory **45**
4.1 Improving transparency overlays 46
4.2 Counter-based detection 49
4.2.1 A counter-based detection protocol 50
4.2.2 Keyless SSL 51
4.3 Commitment-based detection 56
4.3.1 A commitment-based protocol 57
4.3.2 Modifying the ISO-IEC 9798-3-3 protocol 59
4.4 Concluding remarks 63

5 The TAMARIN Prover **65**
5.1 Term rewriting 65
5.2 Multiset rewriting in TAMARIN 68
5.2.1 Facts 69
5.2.2 Rewriting rules 70
5.2.3 Execution 73
5.3 Trace properties 74
5.4 Constraint solving 77
5.4.1 Dependency graphs 77
5.4.2 Guarded trace properties 79
5.4.3 Constraint reduction 81
5.5 Implementation and precomputed sources 83

6 Improving automated analysis of stateful protocols **87**
6.1 Improving heuristics 88
6.1.1 Limitations of the smart heuristic 88
6.1.2 Heuristic locality and annotations 90
6.1.3 Syntax description 92
6.1.4 Tailoring the heuristic 95

6.2 Loop invariants 97
6.2.1 Trace induction and limitations 98
6.2.2 Improving precomputation for handling invariants 104
6.2.3 Reducing irrelevant goals from induction 110
6.3 Concluding remarks 112

7 Case studies **115**
7.1 Analyzing our detection protocols 115
7.1.1 Formal analysis of the Counter protocol 116
7.1.2 Formal analysis of modified Keyless SSL 119
7.1.3 Formal analysis of the Commitment protocol 121
7.1.4 Formal analysis of modified 9798-3-3 123
7.2 Real-world protocols 125
7.2.1 Analyzing WireGuard 125
7.2.2 Analyzing DNP3 Secure Authentication 129

8 Conclusion **139**
8.1 Future directions 140
8.1.1 Auditable wiretapping 140
8.1.2 Further improvements to TAMARIN 141

A Full symbolic models **157**
A.1 WireGuard 158
A.1.1 Model with invariants in premises 158
A.1.2 Model with sources lemmas 161
A.2 DNP3 165
A.3 Counter-based detection protocols 176
A.3.1 Example protocol 176
A.3.2 Modified Keyless SSL 179
A.4 Commitment-based detection protocols 182
A.4.1 Example protocol 182
A.4.2 Modified ISO-IEC 9798-3-3 protocol 186

List of Figures

2.1 A simple example of a protocol. 10

3.1 Message sequence chart of Example 2 24
3.2 Message sequence chart of Example 3 25
3.3 Types of detection 42

4.1 Interactions in Certificate Transparency 47
4.2 A counter-based detection protocol 52
4.3 The modified Keyless SSL protocol 55
4.4 A commitment-based detection protocol 58
4.5 The modified ISO-IEC 9798-3-3 protocol 61

5.1 An execution of the protocol P_{ex}. 75
5.2 An example of a dependency graph 78
5.3 Examples of constraint reduction rules in $\rightsquigarrow_{R,E}$ 83

6.1 Rule colouring annotations 94
6.2 Constraint reduction steps performed in Example 26 99
6.3 Constraint reduction rules added for the last trace atom. 100
6.4 Meier's trace property transformations $\textsc{bc}(\varphi)$ and $\textsc{ih}(\varphi)$. 101

7.1 1-RTT handshake between the initiator and responder. 126
7.2 Benchmarks of several variants of the WireGuard model 130
7.3 The keying relationships between sub-protocols in DNP3:SAv5 132
7.4 The Session Key Update Protocol in DNP3:SAv5 133
7.5 DNP3 rules and invariants 136

Chapter 1

Introduction

Most secure systems depend on secrets in some form, whether they be cryptographic keys, passwords, or other confidential information. These secrets are necessary for the systems to provide the security properties expected of them, and yet it is impossible to ensure *a priori* that the parties who must know the secret will make use of it 'honestly'. Even if it were, it is also impossible to guarantee that classical information has not been copied or is otherwise usable by untrusted (and possibly malicious) parties. Consequently, secret misuse is a serious threat to security.

Since the first uses of secrets for authentication, there have been many technical and procedural measures developed in response to the threat of compromise. Some are designed to prevent secrets from being compromised, like the orders to burn codebooks given to officers in 20th century warfare [4], or the modern day hardware security module [76]. Access control systems, including information classification [29] and the two-man rule [3], are designed to minimize the immediate effect of compromise by limiting the access granted by compromising an individual secret. Once a secret is known to be compromised, or the party holding it can no longer be trusted, revocation mechanisms are designed to limit ongoing damage from the compromise by preventing further use of the secret as swiftly as possible—whether that be by physically replacing locks, or through the technical measures used for certificate revocation on the internet [51, 70].

In addition, there are measures to aid in *detecting* that a secret has been compromised, or that one of its holders is no longer trustworthy. Broadly, this can be done both factually—for example, by observing anomalous or contradictory behaviour—and counterfactually—as with so-called "barium-meal tests", where making use of a

compromised secret leads to some observable activity which should not occur [96]. For the most part, these measures aim to provide only a strong suspicion that compromise has occurred, with the final judgment left to human observers. But, with modern security measures increasingly focused on verifiable formal properties, this hints at a deeper question.

> **Question 1: How can we be *certain* that a secret has been compromised or one of its holders is no longer trustworthy? In other words: when can we *automatically* invoke a response mechanism like key revocation in this setting?**

How we can answer this question depends on the assumptions that we make about how protocols are executed and the network they operate over; for example, if we assume that some agent can forge cryptographic signatures without the corresponding secret, then we would not be able to use such signatures as evidence of secret compromised. In this thesis will make use of standard assumptions from symbolic verification, including that an adversary controls intercepts all messages, and the cryptography used cannot be subverted. We will discuss these assumptions in more detail in Chapter 2.

There exist some systems which aim to provide specific guarantees related to detection. For example, Certificate Transparency [59] and related 'transparency overlays' [9, 10, 19, 98, 99] try to ensure that all participants have consistent information about particular authenticated activities; in the case of Certificate Transparency, which certificates have been signed off by a trusted authority. If they have consistent information, then participants could collectively audit these activities, ensuring that, for example, no certificates have been signed without a request by the appropriate party. But there are few examples where formal verification has been applied to ensure that these systems actually meet their design goals, and even those which have been analyzed are typically verified only for small examples. Even transparency overlays, with their stated goal of guaranteeing detection, shy away from prescribing action and instead assume human intervention.

If we wish to be confident enough to declare a secret compromised and invoke response mechanisms automatically, formal verification is a necessity. Early systems, like the clone detection mechanism in the ANSI-41 [1] specification, were intended to allow automatic revocation but were never formally analyzed to ensure that detection was correct. And indeed, in the case of ANSI-41, it was discovered soon after implementation that the mechanism frequently generated false positives, rendering it

useless [95].

Much of the difficulty in performing formal analysis of systems like transparency overlays is found in their complexity. There are many roles involved, each of which is tasked with maintaining state across multiple protocol runs and communicating with many different agents. As we will see later, answering our first research question, naturally leads us to develop new detection protocols that enable automatic response. However, as we will also show, these protocols are necessarily stateful, which is a challenge for automatic protocol analysis tools. This motivates our second research question.

> **Question 2: Can we improve our ability to formally analyze stateful protocols?**

We answer this question in the affirmative, through careful consideration and augmentation of an existing protocol analysis tool that is well-suited for this purpose. The class of stateful protocols contains much more than just detection protocols, which motivates our final research question.

> **Question 3: Can we use our work to perform novel analysis for both protocols which detect secret misuse, as well as complex real-world protocols for which no formal analysis currently exists?**

Through these three questions, we aim to broadly increase the assurance provided by security protocols. We will construct the foundations and theory for protocol designers to build and improve on protocol designs that allow for detection of misuse—and will give concrete examples where we have done so. We will improve a tool widely used in protocol verification so that it is easier to model and analyze complex protocols. Finally, we will apply these improvements to formally verify not only our own protocols, but also to analyze real-world protocols and contribute to their ongoing development.

1.1 Approach

Each of our research questions requires a different approach. Here, we give intuition for how we will proceed before we discuss our contributions.

Question 1 Distributed programs run by communicating agents to achieve security goals within a particular threat model are broadly captured by the notion of security protocols. Thus it is natural to consider designs to automatically detect and respond to key compromise within the field of protocol design and formal analysis.

Though we began by discussing secret compromise, detecting compromise itself is not possible in general. If an attacker simply learns a secret but never makes any use of it, then there is not necessarily any evidence to conclude that compromise ever occurred. However, in many cases—particularly when a secret is used for authentication—the attacker has some other goal which they can only perform using the secret; for example, to log into a service, to request a document, or to trigger a specific action of the system like opening a door. Additionally, a malicious agent may abuse their knowledge of a secret to subvert a protocol's security without a specific notion of compromise. For this reason, we refine our terminology and refer to secret *misuse*, by which we mean *any* use of a secret to perform some action that would not occur if all participants were honestly following the protocol.

There exists prior work on specific protocols in certain domains to make secret misuse discoverable, but nevertheless there has been no attempt at generalizing these protocols or establishing the foundations of their operation. Further, even within existing protocols there are few examples where formal analysis has been applied to verify these 'detection' or 'discoverability' properties.

To answer this question, we begin by reviewing background on security protocols and prior work. Certain common threads in this prior work will give us some intuition to construct motivating examples. From there, we can define a model of protocol execution that is as general as possible while allowing us to prove certain restrictions on the ability to detect. Thus we can construct a categorization of detection into distinct types, and examine the implications of these restrictions.

Question 2 There are several protocol analysis tools which allow state to be modelled to some degree. Many of these, like the original abstraction used by ProVerif [12], only allow for *monotonic* state—state where no values can be removed or changed. Others, like the AVISPA tool [6], require concrete bounds on the number of sessions and fresh values generated.

Two main tools have seen substantial success at performing automatic analysis of stateful protocols: StatVerif, described in [5], and the TAMARIN prover described in [64] (which also underlies SAPIC [55]). StatVerif extends the process calculus of

ProVerif with explicit state constructions, allowing it to model non-monotonic state. Tamarin takes a different approach, with protocols modelled as a multiset rewriting system that transforms global state; non-monotonic state is thus captured as a natural consequence of the protocol execution model. SAPIC compiles specifications written in a stateful process calculus to Tamarin input.

Our goal, beyond proving our own constructions, is to improve stateful protocol verification for use on real-world protocols. While StatVerif has been used to perform analysis of a toy model of a security device and a contract signing protocol [78], Tamarin has featured in verification efforts of TLS 1.3 [27], V2X revocation protocols [93], PKCS#11 [57], flow integrity in industrial control systems [38], and several other real-world applications [86]. Thus, we focus our efforts in answering our second research question towards the Tamarin prover.

We will first examine the background behind the constraint solving algorithm and protocol execution model in Tamarin, and then proceed by improving two specific aspects of the implementation that limit its ability to automatically analyze stateful protocols.

Question 3 For our third question, we begin by applying our improvements to successfully analyze the detection protocols that we design. We then look towards complex real-world protocols without prior analysis. First, we apply our work to analyze WireGuard [35], an increasingly popular VPN protocol. Through our analysis, we prove several strong security properties in our model. Additionally, our discussions with the WireGuard team lead to a protocol change which decreases the complexity of the protocol with no loss in security; the change is also upstreamed to the Noise protocol framework [73] used by other projects. The reduced complexity dramatically simplifies formal analysis of the protocol, and the upstream change will allow other projects to be analyzed more easily in the future.

We also examine an extremely complex real-world protocol, the DNP3 Secure Authentication protocol [39] used to secure utility grids. For a variety of reasons, this protocol has resisted any prior attempts at formal analysis. We discuss its complexity and the considerations necessary to model it, and successfully verify its security properties automatically.

1.2 Contributions

Our contributions span theoretical results that motivate and inform protocol design, developments that allow these designs to be realized and verified, and practical improvements and analysis of real-world protocols that are widely used today.

We answer our first research question by providing the first general foundations for the automatic detection of the misuse of secrets for automatic response. Our focus on detection as a verifiable security property with the requirement of no false positives ensure that our developments can be used to automatically revoke keys, access, or invoke other countermeasures; our foundational approach also provides new insights into the design choices in existing systems designed to detect. We identify several design principles relevant to the design of detecting protocols, and propose an improvement to existing systems as well as several two-party protocols capable of detecting misuse in practical scenarios.

Our second research question is answered through a series of developments to Tamarin, improving several aspects of its heuristics and computation steps to make it faster and easier to perform automated analysis of stateful protocols.

The improvements we make to Tamarin allow for the analysis of protocols that previously seemed intractable. We use these improvements to verify several properties of our proposed detection protocols. Finally, we demonstrate the power of our work on Tamarin by performing the first formal analysis of both the WireGuard handshake protocol and the complex DNP3 Secure Authentication protocol.

1.2.1 Originality and other contributions

The foundations of detection and related protocols originally appeared in [67] as joint work with Cas Cremers, Mark Ryan, and Jiangshan Yu, who contributed to the motivating examples and protocol designs that appear in that paper; these have been modified and expanded in this thesis.

Our work formally analyzing the DNP3 Secure Authentication protocol appears in [26] along with detailed discussion of the context and specific security properties of the standard by Cas Cremers and Martin Dehnel-Wild. In this thesis, we focus specifically on the formal modelling and verification, which is entirely our own work.

The analysis of WireGuard was done as joint work with its creator, Jason Donenfeld, who ensured that the protocol model accurately reflected WireGuard's design and

proposed the modification to NoiseIK based on our work; our analysis can be found in [36].

In addition to the work that appears in this thesis, we contributed to the design and symbolic verification of ART [25], a protocol to provide post-compromise security for end-to-end encrypted group messaging.

1.3 Overview

We begin by answering our first research question, with our first three chapters focused on developing and applying a theory of detection. Chapter 2 reviews background on security protocols and protocol models, as well as prior approaches to mitigating or detecting secret compromise within protocols. In Chapter 3, we develop the foundations for the detection of key misuse in general, proving necessary conditions to prevent false positives and providing a categorization of the ways in which misuse can be observed. In Chapter 4, we show how these foundations allow us to consider an existing system for detection in a new light, and propose several detection protocols that we will later verify.

Next, we move to answering our second question, with a view towards the Tamarin prover. In Chapter 5 we provide background on the theory and implementation of Tamarin. We then, in Chapter 6, develop several improvements to Tamarin which make it even more suited for the analysis of detection protocols, and show two case studies for our improvements.

In Chapter 7, we tackle our third question. We formally analyze the detection properties proposed in Chapter 4, and show the broader applicability of our improvements to Tamarin by analyzing two well-known, real-world protocols with no prior formal analysis.

Finally, we conclude in Chapter 8 with a summary of our work and future directions for research.

Chapter 2

Background and related work

This chapter presents background on security protocols and ways secret compromise has been considered in protocol design in prior work. We begin with an overview of protocols and models in Section 2.1, followed by examples of prior work on protocols with the goal of restoring security properties after compromise in Section 2.2. Finally, in Section 2.3 we examine several examples of related work detecting secret misuse in specific domains.

2.1 Security protocols

A security protocol is a program distributed across multiple agents, comprising both local operations by that agent and message exchanges between agents. Security protocols are designed in order to guarantee properties of the execution of the protocol, even in the presence of an attacker. For example, we may wish for a protocol to have the property that, after two agents (say Alice and Bob) execute the protocol with each other, Alice can be sure that Bob intended to send her precisely the data she received. Or, for example, that at no point could any other party learn the data sent between the two of them. Such properties are called *security properties*.

An example of a security protocol is shown in Figure 2.1. We write $\{\!|\ |\!\}_{\mathsf{sk}(R)}$, to indicate a message concatenated with a signature computed using the key in the subscript, i.e. both the contents and an authentication tag are sent in m_2. This protocol has two roles: the initiator I and the responder R, each of which prescribes actions for an agent to execute. An agent Alice executing the role I of this protocol would begin by generating a random value ni before sending it over the network to an

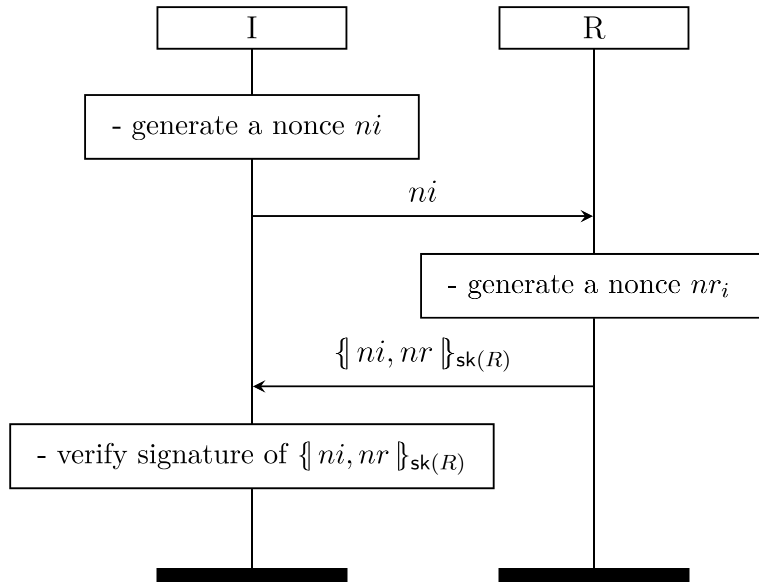

Figure 2.1: A simple example of a protocol.

agent Bob executing the role R. Bob would receive a *ni* from the network, generate a new random value *nr*, and then sign both *ni* and *nr* with his secret key and send it back to Alice. Once Alice receives this message, she can verify the signature with the public key belonging to Bob, and either successfully complete the protocol if the signature is correct, or fail before proceeding with any further execution. A complete instance of a protocol role executed by an agent is called a *session*.

Although the protocol is described in terms of messages sent directly between the two agents, this is rarely how systems actually work. The goal of a security protocol is to guarantee some property even in an adversarial setting—for example, when messages are sent over an open and shared network that an adversary can interact with. The assumptions that are made about the adversary's capability depend on the setting and type of analysis; in this thesis we will take a symbolic approach based on the Dolev-Yao model [34].

2.1.1 The Dolev-Yao model

To analyze a security protocol and verify its properties, it is necessary to make some assumptions about the environment it operates in. After all, it is trivial to guarantee that a message is from a particular agent if one assumes that no other agent can send

messages.

When considering security properties, we would like a protocol to provide security in the most hostile environment possible—against the most powerful adversary that is still relevant. The agents running the protocol may not expect such an adversary, but it is better to be safe than sorry. Furthermore, it is easier to reason about potential attacks without multiple, intersecting restrictions on adversary actions.

In 1983, Danny Dolev and Andrew C. Yao published a manuscript introducing formal models for analyzing security protocols [34]. What is now called the *Dolev-Yao* model has evolved from their work, and is captured by three main assumptions:

1. **(Perfect cryptography)** Neither the adversary nor any agents know how to subvert cryptographic operations. Specifically, cryptography can be modelled by a term algebra, in which terms must be derived from operations on other terms. Thus, analysis can be done *symbolically*.

2. **(Unbounded execution)** The protocol can be executed an arbitrary number of times by arbitrary sets of agents.

3. **(Network adversary)** The adversary is the network: all messages are sent to the adversary, and the adversary can deliver any message that they can derive from everything previously sent on the network. This includes messages from previous protocol runs between any agents on the network.

For the protocol in Figure 2.1, in the Dolev-Yao model we assume that despite being able to intercept ni, the adversary cannot compute a signature using $\mathsf{sk}(R)$ without knowing $\mathsf{sk}(R)$. Thus, if $\mathsf{sk}(R)$ is secure then upon completing the protocol, the agent executing the role I can be certain that m_2 was constructed by R after ni was generated. The agent executing the role R, however, has no guarantee about the provenance of ni, as an adversary could have generated it themselves (or redirected a message from any other agent).

2.1.2 Compromising adversaries

In the original Dolev-Yao setting, security properties are considered in the context of protocol runs, under assumptions that for example certain keys are secret from the adversary. There are many desirable security properties that do not fit within this limited scope, however: protocol participants may wish that their communication is

guaranteed secret even if an adversary compromises a particular key *after* they run the protocol, for example.

To consider more complex security properties like these requires increasing the capabilities of the Dolev-Yao adversary. We augment the adversary with the ability to compromise particular secret terms held by the protocol participants, thus allowing us to reason about security properties based on when this compromise occurred, and for what terms. Well-known properties that require a notion of compromise include

1. **Perfect Forward Secrecy:** some shared data (typically a session key) derived by running the protocol remains secret from the adversary, *even if they compromise the long-term keys of the participants afterwards*;

2. **Key Compromise Impersonation (KCI) Resilience:** an adversary cannot execute the protocol as an agent A *even if they compromise the long-term keys of all other agents*.

We will make use of similar adversary assumptions in our model, because a notion of compromise is integral to the meaning of detection without false positives. For a full treatment of compromising adversaries in a symbolic model, see the work of Basin and Cremers in [7].

2.1.3 Stateful protocols

In addition to augmenting the power of the adversary, we also consider additional capabilities of the agents executing a protocol. Nearly all security protocols require some notion of state; at its most basic, an agent must know what step of the protocol to execute next, or store a nonce generated during a session.

In early protocols and protocol modelling, state was considered only in the context of a single session. This makes some reasoning much easier; it means past executions of a protocol have no impact on the current execution. More generally, it guarantees that protocol executions can be performed in parallel by agents, with no causal dependencies between them.

Many modern protocols, however, make use of state that spans several sessions, modifying it as they go. Indeed, certain strong security properties require state; for example, keeping state is a necessary condition for a protocol to achieve post-compromise security [24]. Post-compromise security is the property that compromise of an agent's key by an adversary is 'healed' by any later session without adversary

interference, in the sense that the adversary no longer gains any advantage from knowing the prior key. This requires that, after each session, agents must hold some new secret data that the adversary cannot derive using the previously compromised key (and so state to store that secret material). Thus, we assume stateful agents in our work, and make use of a protocol analysis tool, TAMARIN, which allows for them.

2.2 Restoring security after compromise

In a number of domains, there have been efforts to minimize the effects of secret compromise through restoring the security of protocol sessions following a compromise event. Note that this stands in contrast to, e.g. perfect forward secrecy, which is a security property of the sessions that occurred prior to compromise.

Here, we review some approaches to security properties after compromise. Note that many of these are domain specific and provide only informal guarantees, with the exception of the work by Cohn-Gordon *et al.* in [24].

2.2.1 Key-evolving cryptosystems

Key-evolving cryptosystems (e.g. [46, 48], for a survey see [44]) have been proposed to mitigate the damage caused by compromised secret keys, through the use of periodic key changes. In the symmetric setting, a sender and a receiver share an initial long-term secret which is used to derive a set of keys that is only valid for a certain time period. In the asymmetric setting, each party holds the public keys of the other, and updates this stored key when they receive an authenticated 'key evolution' message. Thus, a compromised agent key is only valid until the next key evolution by that agent.

Key-evolving cryptosystems help limit the potential damage of one-time key compromise, but have limited effectiveness against a network adversary. Key evolution can be triggered by an adversary after compromise, and that adversary could continue to intercept and modify messages from the compromised agent indefinitely. In some systems, the choice of a new key is arbitrary, in which case the adversary can return to a passive role simply by evolving the key back to the one held by the real agent.

2.2.2 TPM authentication protocols

A Trusted Platform Module (TPM) [90] is a hardware component designed so that platforms can provide better protection of sensitive data. It does this by storing keys in its shielded memory, allowing access and use of these keys only through a specific interface.

TPMs provide two kinds of key-use authorization protocols: the Object Independent Authorisation Protocol (OIAP) and the Object Specific Authorisation Protocol (OSAP) [30]. These protocols are the interface through which a user process can, for example, authorize key activity on the TPM. The accessing user process is authenticated by the TPM in part with a series of 'rolling nonces' which include the nonces generated in the *previous* access by both the user process and the TPM. This acts as a sort of evolving key constructed from past sessions.

The rolling nonces are intended to maintain freshness in every message, but they have a side effect of ensuring the participating agents were also involved in the previous session. This construction is not completely resilient against key compromise, though: an adversary who can inject messages can make use of the TPM temporarily, then 'resynchronize' the nonces between the user process and TPM.

2.2.3 Key ratcheting and post-compromise security

When previous keys are combined with new key material, and that new material is generated so that it has perfect forward secrecy, the result can provide much stronger security properties than a simple key update scheme.

The Signal protocol makes use of a 'key ratcheting' process called the Double Ratchet Algorithm [68], designed for messaging systems. In this protocol, agents periodically perform Diffie-Hellman exchanges as they send and receive messages. When one of these key exchanges occurs, they derive a shared key from the combination of both the previous key and the result of the key exchange. This is referred to as 'ratcheting' the previous key, in reference to physical ratchets which can only be turned in one direction.

In [24], Cohn-Gordon *et al.* introduce *post-compromise security*: a security property that provides formal secrecy guarantees for communication after compromise. This is accomplished through key ratcheting, like the Double Ratchet Algorithm. Their work is notable for focusing on formally stated security properties and analysis, in contrast

with the other approaches reviewed here (though Signal has been recently analyzed by Cohn-Gordon *et al.* in [23])

2.3 Detection of misuse

In several domains there have been attempts to develop protocols which allow for the detection of secret misuse. The omnipresent problem of secret compromise—and therefore the usefulness of detecting—has lead to parallel developments in different domains, each with their own terminology. Here, we review past work on 'clone detection' from the field of embedded systems and 'fault detection' from the field of component systems. We also discuss existing work on transparency overlays, a class of protocol designs with the goal of letting participants audit particular uses of secrets.

2.3.1 Clone detection methods

In some domains there have been efforts to detect compromise through some predictable constraint applied to agent behaviour (like a persistent, increasing counter value), so that adversarial activity will lead to a violation of the constraint. This has primarily been applied for detecting cloned physical devices, but more recently has been used more broadly to ensure the consistency of public logs. Note that key rotation mechanisms as described above can also cause adversary activity to contradict honest behaviour (one of the two may change their key while the other continues to use an old one), but the mechanisms described above do not attempt to detect this.

When physical devices contain secrets that are intended to uniquely authenticate the device, compromise of device may allow an adversary to authenticate as if they had the device even after their physical access is revoked. This type of attack is called cloning, and is a common problem with portable devices where temporary physical access can allow for attacks on the device hardware. In some domains, protocol-based solutions to cloning have been proposed, which we briefly review below.

ANSI-41

One of the early standards for authentication in North American telecommunications, ANSI-41 [1], introduced a feature for clone detection in the form of a 6 bit `CallHistoryCount` variable. The value of this counter was included by the device as part of its authentication to the network, and would be incremented by both the

device and the service provider independently whenever the phone began a call [95]. If a device were cloned, then a phone call made by one of the two devices would cause the other to disagree with the network about how many calls had been made.

Unfortunately, the unreliability of early mobile networks meant the counters could often get desynchronized even without cloning. As such, the protocol specification requires that if the carrier wishes to take automatic action, it should only occur "if the stored count and the `CallHistoryCount` do not significantly match" [1]. In general the unreliability of synchronization prevented carriers from risking automated response, but nevertheless large discrepancies in call history counts, especially repeated ones, were useful for network operators to determine if a device was likely to be cloned.

Subsequent authentication standards used for mobile phone networks abandoned a protocol-level clone detection mechanism in favour of hardware security for keys to make cloning somewhat more difficult [97], though it is still possible [61, 75].

RFID tag cloning detection

Mechanisms [13, 60, 100, 101] for detecting cloned RFID tags in the supply chain have been widely studied. RFID tags are a particularly challenging environment, as a cryptographically secure protocol and hardware security features would typically be too expensive to deploy.

Protocol-based solutions to cloning in this domain rely on detecting conflicting information. For example, Zanetti *et al.* [101] give a solution in which RFID readers write random values to RFID tags as they pass through the supply chain so that the tag accumulates a sequence of random values. Cloned tags are then detected by observing contradicting sequences for the same tag identity.

This solution points to the intuition that detection of a violation of some property *a posteriori* based on contradictory observations can be much more lightweight than a protocol which achieves some security property in each session. In this case, we detect that the tag identity has been in fact identifying that particular tag uniquely (with known probability), rather than having each session securely authenticate the tag.

Methods in wireless sensor networks

A different approach to clone detection was proposed in [14]. The authors propose the generation of a large set of pre-shared keys, out of which individual nodes select a subset of constant size. Links in the sensor network are established between any two

nodes which share a key in common, and these direct links are used to establish keys with every other node in the network. An adversary may compromise a particular key or all keys used by a particular node. The protocol then detects the addition of cloned nodes through examination of the statistics of key usages by nodes. Because the pre-distributed keys were selected from a larger pool, and the number of nodes is known, the distribution of keys can be determined by any node in the network; further, each individual node knows the keys which it uses to communicate. To determine the combined statistics of key usage, each node forwards its key usage to a central authority, and if there are anomalies the central authority determines the anomalous key usages and sends a message to revoke them across all nodes.

The complexity and specificity of this method is driven in large part by the many requirements specific to wireless sensor networks. The high power requirements of public key cryptography require the pre-establishment of symmetric keys across nodes. Wireless sensor networks also tend to involve many individual nodes, allowing for statistical guarantees about both incorrect key revocation and whether it will adversely affect the connectivity of the network.

2.3.2 Causality-based fault detection

In [45], Gössler and Le Métayer investigate causal connections between events in a symbolic trace. If correct behaviour leads to particular causal relationships, then faulty systems can be determined by their violation of causality. Specifically, when a system performs an action without some associated triggering action prior, the behaviour must have been faulty.

This work focuses on causality violations as a basis for determining system faults, but the concept of determining misbehaviour counter-factually is a useful concept in general. With the existence of cryptographic authentication, causal structures in protocols can be made very precise, even in settings where secrets may be compromised: a message from an agent B to another agent C, for example, may not only be authenticated by B's key but could also include an artefact authenticated by another agent A. Thus, even in traces where B's key is compromised, an adversary may not be able to construct the message without a preceding action by A.

We formally define and discuss a notion of causality as part of our work in Chapter 3, as it defines the most powerful of the three categories of detectable misuse in our complete classification.

2.3.3 Transparency overlays

Transparency overlays are systems for auditable public logs, inspired by the need to detect compromised Certificate Authorities (CAs) in order to address some of the problems with the current trust model used for securing the internet (for a general review of problems and earlier research not specific to detection, see [21]). Proposals include Certificate Transparency (CT) [59], Enhanced CT [77], Accountable Key Infrastructure (AKI) [50], PoliCert [84] (an extension of AKI), and Attack Resilient Public-Key Infrastructure (ARPKI) [9].

Though these proposals differ substantially in practical implementation details, their ability to aid in detection relies on similar fundamental structure and insights—primarily, that misuse is defined by a CA's key authenticating a certificate for a domain without authorization by that domain's owner. Thus, detection in general requires that all uses of a CA's key for every domain must be able to be audited and verified by those domain owners. Detection is provided through the use of one or more independent entities which act as logs of all certificates issued, where the completeness of the log is ensured by enforcing that any certificate is valid only if it is in the log.

The detection of misissued certificates relies on the requirement that a certificate for a domain is only correctly issued when the owner of the domain *causes* it to be, by requesting it. As such, a domain owner can identify any certificates issued for their domain that should not have been, in a similar way to causality-based fault detection above. We expand on transparency overlay systems specifically in Section 4.1.

These systems rely on the log server presenting the same information to all participants, so that all agents involved can be certain that the information they see in the log is the same information that has been audited by other agents. In lieu of a mechanism by which this could be guaranteed directly, transparency overlays instead rely on detecting if a log server's key was used to authenticate two inconsistent views of the log.

This detection works by requiring all log views to be in some one-way fashion derived from all other views presented by the same server, so that it is computationally infeasible to return to an earlier log state. In the process of developing our foundations in Chapter 3 we will show why this is necessary.

Notably, ARPKI [9] makes use of formal verification to prove specific security guarantees. A formal symbolic model of the protocol was developed alongside the protocol itself. The model allowed the authors to verify specific security properties of

the protocol, and makes the conditions which imply those properties explicit. The co-development of a symbolic model also requires that the protocol itself be completely (symbolically) specified.

2.3.4 Accountability and verifiability

Verifiability, originally considered in the context of e-voting protocols is focused on the perspective of a particular agent (referred to as a judge) deciding whether, given some information from a protocol execution, they should accept it or not (i.e. whether it has some property the judge wishes to enforce). Early e-voting work considered forms of verifiability termed *individual verifiability* and *universal verifiability* [79], in which the goal of the judge is respectively to determine whether their ballot was counted or to determine if all ballots shown on some bulletin board were counted. This was later expanded to *eligibility verification* [56], in which the judge also determines that all counted votes are *eligible*: no individual voter cast more than one ballot.

Expanding on these definitions, Küsters, Truderung, and Vogt propose formalized definitions of verifiability as well as a notion of *accountability* in [58]. Informally, accountability also requires that if some malicious activity occurs so that the judge does not accept the protocol execution, the judge can place blame on at least a subset of the agents who misbehaved (and never blames honest participants).

This is broadly similar to what we wish to accomplish with misuse detection, but has conceptual differences to misuse detection. Specifically, accountability focuses on misbehaving parties rather than a compromising adversary. While a compromised party can readily be thought of as a special kind of misbehaving agent, we will see that the ability to differentiate the state of an adversary from the state of a compromised party is fundamental to allow for two-party protocols that detect misuse. Additionally, the definition of accountability does not naturally capture a notion of ongoing detection performed across sessions with shared state; judging happens as a phase after protocol execution. Thus, detection of misuse takes a different perspective from accountability that is more intuitive for constructions like transparency overlays and the two-party protocols in Chapter 4.

Chapter 3

Foundations of detection

In this chapter, we develop formal foundations and explore the design space for the sound detection of secret misuse. Our formalization enables us to precisely define the concepts relevant to detection, and to explore the design space systematically. We will use the resulting foundations and intuition to improve existing systems as well as design new protocols in Chapter 4.

Within this chapter, we begin by discussing motivating examples in Section 3.1, which we will return to later in the chapter. We then introduce some notation and terminology in Section 3.2 before developing our formal results. We use this to examine in Section 3.3 what it means to be able to detect without false positives, and categorize the ways in which that can occur, with varying requirements on the agents involved. Finally, we end the chapter by discussing the implications for designing protocols which can detect misuse, with design principles to consider.

3.1 Motivating examples

In order to investigate how we could detect compromise to allow automatic response, we first consider some motivating examples. We observe that if an attacker silently obtains a secret but performs no visible actions based on this information, the compromise fundamentally cannot be detected; classical information is cloneable and so there is no information-theoretic consequence of an attacker learning it. Furthermore, if the attacker obtains all necessary secrets to impersonate the original owner, performs actions using those secrets that are identical to the expected behaviour of the original owner, *and* the original owner performs no further actions (e.g., because they are

deceased), then to all other participants the attacker's behaviour must be indistinguishable from the original owner. In a way, the attacker would have completely taken over the life of the original owner. Thus, informally, the only situation in which we can hope to detect the misuse of those compromised secrets is when the attacker deviates—or rather, is forced to deviate—from the original owner's behaviour or ongoing actions, either because the dishonest behaviour is inherently and noticeably different or the ongoing actions create discord.

As we are interested in protocols which allow automatic response, participants must be able to logically conclude that some deviation *must* be the result of misuse, in order to ensure there are no false positives. This is notably different from the field of *anomaly detection*, which also seeks to identify deviation from 'honest' actions, but does so probabilistically. The challenge in anomaly detection is generally in identifying actions which are *allowed* but *very unlikely* to be part of normal activity. Unfortunately this often leads to false positives, and as such typically requires human oversight or only minor responses (e.g. requiring a user to re-enter their password).

Consider the following examples of protocols and attacks which allow agents to differentiate adversary action from action by the honest agents, which each examine a different aspect of detection that we will return to in Section 3.3.1.

Example 1. Alice has a secret $\mathsf{sk}(A)$ which she can use to authenticate messages. The adversary compromises this secret, and sends an authenticated message which is obviously incorrect. For example, the authenticated message might be "I compromised this secret".

Example 1 is unlikely to occur in practice, but it is still a valid action the attacker could take so it is important to take it into account.

Example 2. Alice and Bob have signing keys $\mathsf{sk}(A)$ and $\mathsf{sk}(B)$ respectively, and send each other messages authenticated with their keys over a public channel. They each maintain a counter, and when Alice sends a message to Bob on the i-th session, she increments her counter, generates a new nonce na_i and includes them both in her message along with the last nonce received from Bob (nb_{i-1}). Upon receiving this message Bob checks that his last nonce matches, increments his counter, and checks that it matches the one in the message. Similarly, when Bob sends a message to Alice, he includes a newly generated nonce nb_i, his counter value cb_i, and Alice's last nonce na_i. The next message from Alice contains a new nonce na_{i+1}, nb, and an incremented

counter value ca_{i+1}, the next message from Bob a nonce nb_{i+1}, na_{i+1}, and his counter value cb_{i+1}, etc. This protocol is shown in Figure 3.1

Example 2 illustrates a simple case in which misuse can be detected. If an attacker gains knowledge of $\mathsf{sk}(A)$ and the current value of the counter, and injects a new message purporting to be from Alice, then Alice's and Bob's value of the counter will become de-synchronized and they could detect upon comparing these values that $\mathsf{sk}(A)$ was misused. However, this is somewhat limited, as an attacker with knowledge of both keys who observes a counter value could strike up conversations with Bob, then wait for Alice to send messages. By intercepting these and returning a message to Alice which appears to be from Bob the adversary can increment Alice's counter until it matches, and then inject one more message to each to resynchronize their nonces. Alice and Bob are left in a state as if the attacker were never involved.

Note that because Alice and Bob's counter values rely only on the number of messages exchanged and not on their content, it is impossible to determine if they agreed on all previous message content. Thus, the attacker can resynchronize them even after they have disagreed about the messages exchanged.

Example 3. Instead of using a counter, Alice and Bob adopt a system of 'rolling nonces with hash chains'. When Alice sends her authenticated message to Bob in the i-th session, she includes a new nonce na_i and a hash chain of the previous nonces used by both parties in the conversation. Bob then checks the value of the hash chain matches his own, and when sending a message to Alice does likewise, including a new nonce nb_i and extending the hash chain with na_i. The next message from Alice contains a new nonce na_{i+1} and the hash chain extended with nb_i, etc. The i-th session of this protocol is shown in Figure 3.2.

In Example 3, suppose an attacker obtains Alice's key k_A along with the current nonce and hash chain. The attacker can inject conversations with Bob, which necessarily extends Bob's hash chain with new values. If the attacker ever stops intercepting messages between the two, his session will be detected, since the hash chain of Alice will not match and the adversary has no way to 'rewind' Bob's additions to his hash chain. Indeed, even if both keys k_A and k_B are compromised, this example with hash chains allows for detection if ever the attacker tries to back out of the conversation, as any session the attacker carries out with either of them has an irreversible effect on their state.

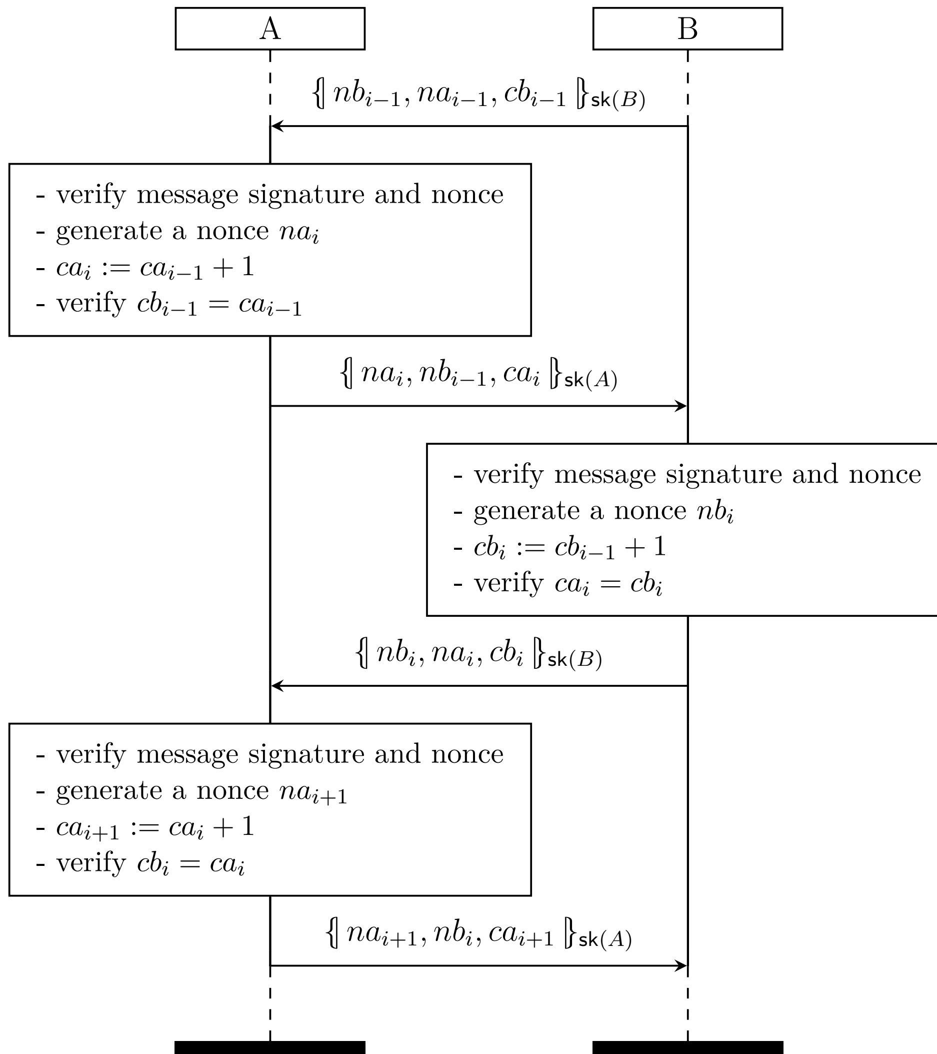

Figure 3.1: The protocol described in Example 2, beginning from the i-th session.

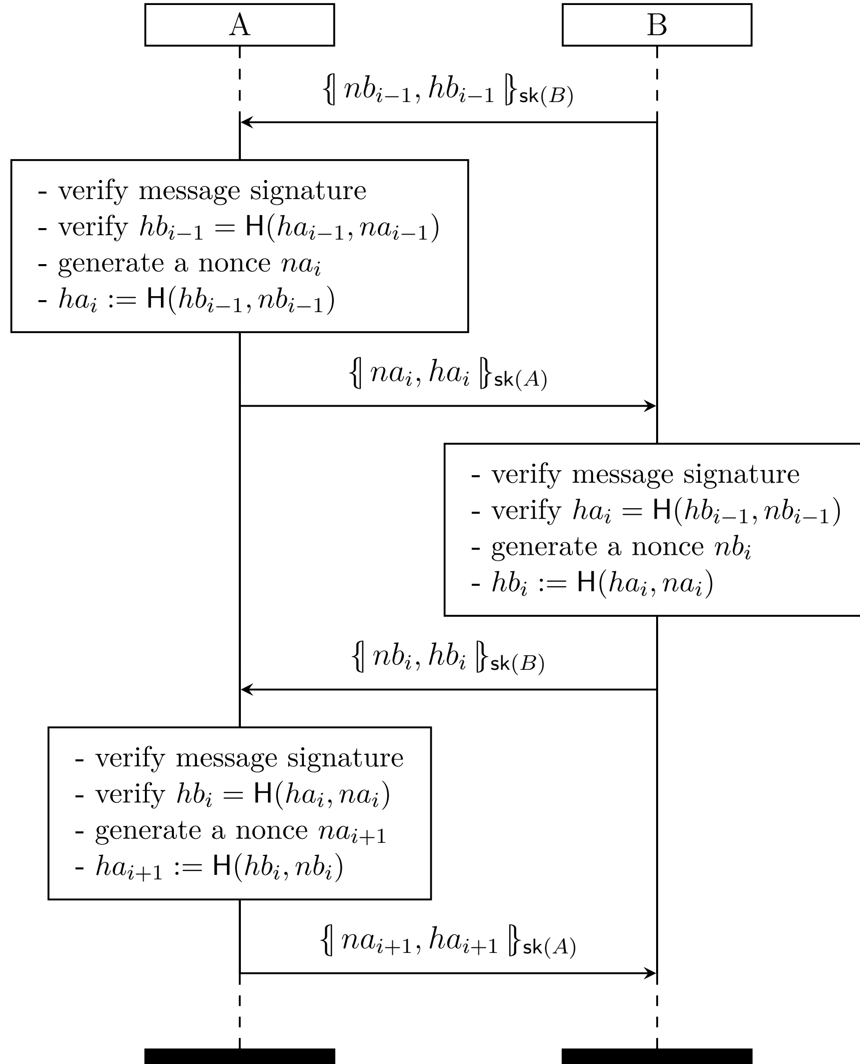

Figure 3.2: The protocol described in Example 3, beginning from the i-th session, assuming a function $\mathsf{H}(x, y)$ which extends a hash chain x with an additional term y.

We will come back to these examples explicitly later on, but they motivate some intuition for how a coupling of information between past and present sessions allows adversary action to be noticed, at least in principle.

3.2 Notation and framework

To build towards some formal descriptions and results, we now introduce some notation, as well as a framework for discussing generic protocol execution with stateful agents. This enables us to formally define the terms we have been using, like what it means to soundly detect compromise, which we will need later when building towards our results. In this section we also show some properties that result in our framework.

3.2.1 Notation

Sequences are used throughout this thesis. Given a sequence s, we address the i-th element as s_i and use $idx(s) = \{1, \ldots, |s|\}$ to refer to the set of indices of s, where $|s|$ is the length of s. We use angle brackets for sequences, where $\langle\rangle$ denotes the empty sequence and $\langle e_1, e_2, \ldots, e_{|s|}\rangle$ is used for the sequence comprising element e_1 followed by e_2 and so on. The concatenation of two sequences s and s' is written $s \cdot s'$. We overload set notation for sequences and write $e \in s$ for a sequence s if and only if $\exists i \,.\, s_i = e$.

In order to discuss a particular subsequence, we use the sequence projection operator. For a sequence l and a set S, the projection $l|_S$ is defined as

$$l|_S = \begin{cases} \langle\rangle & \text{if } l = \langle\rangle \\ \langle l_2, \ldots, l_{|l|}\rangle|_S & \text{if } l_1 \notin S \\ \langle l_1\rangle \cdot (\langle l_2, \ldots, l_{|l|}\rangle|_S) & \text{if } l_1 \in S. \end{cases}$$

Projection is distributive over sets of sequences, so a projection of a set of sequences is the set of each sequence with the projection applied.

3.2.2 Reasoning about protocols

We introduce basic notation for a generic class of protocols and an abstract notion of detection. This enables us to formally define what it means to detect compromise, and what is necessary for detection.

We assume a finite set of agents Agent as participants, each of which has some associated state, access to a random number generator, and which can communicate only through sending and receiving messages on a network. Agents perform actions according to a *protocol*. A protocol is a deterministic algorithm to be run on a Turing machine with agent state as input, which returns an action to perform. Such actions may include accessing an external resource—e.g. sampling the random number generator, or accessing the network to send or receive messages—or internally modifying their state, etc. From this, a transition system arises in which an agent with a particular state performs a specific action dictated by the protocol, which then results in a new agent state depending on the action and potentially the state of the network or the result of sampling the random number generator. We write Protocol to denote the set of all protocols.

More formally, the protocol dictates an action taken as a deterministic function of agent state. The possible states resulting from that action may rely on the state of the network, and may (if sending a message to the network) influence the state of the network. The result cannot depend on the state of other agents or the adversary directly, only the network, and we assume that agents can access the network only through `send` actions, which add a particular message to the network, or `recv` actions, which can receive a copy of any message from the network. In a particular execution, one of these possible states is chosen non-deterministically, and the combination of the action performed, the agent performing it, and the result is called an *event*. For example, we use $\mathtt{recv}_x(m)$ to denote an event in which an agent x performed a `recv` action and received message m (though we will often drop the agent identifier or message if they are not relevant). Each agent x has a transition relation

$$Step_x \subseteq State \times Event \times State,$$

where an agent event fully determines the resulting state transition, i.e.

$$(st, e, st') \in Step_x \wedge (st, e, st'') \in Step_x \Rightarrow st' = st''.$$

We will use these agent events, along with adversary events, as a record of protocol execution.

To model adversarial activity, we assume the existence of an adversary with similar resources to the agents, but with the additional ability to perform actions which

remove messages from the network and compromise parts of agent state. Adversary actions are provided by a deterministic algorithm, which we call an *adversary model*. It runs on a Turing machine, taking adversary state as input and outputting an action for the adversary to perform. We define events for the adversary similarly to agents above, and denote the set of all adversary models *Adv*.

The definitions above do not allow for malicious agent activity, since all agents are assumed to follow the protocol. We emulate malicious agents instead through the adversary model, which allows for the adversary to learn arbitrary terms (and thus all terms held in an agent's state). Since agent actions are a function of their state, and since all communication with other agents occurs through the adversary-controlled network, this is sufficient to allow adversary emulation of an agent. This makes it easier to abstractly distinguish potentially malicious actions from honest and correct events in the trace, while allowing for over-approximation of the abilities of malicious agents (since the adversary model may include controlling the network or compromising additional agents).

Each combination of a protocol $P \in \text{Protocol}$ and adversary model $\mathcal{A} \in \mathit{Adv}$ gives rise to a transition system with agent states, the network as a set of messages, and the state of the adversary. At each step, either an agent or the adversary performs an event, with a corresponding state transition, and possibly a change of network state. We log each adversary or agent event in a sequence called a *trace*. The particular representation of events in the trace is not important for our purposes; instead, we require only that the trace contains sufficient information to reconstruct the state of the adversary and every agent at each point in the trace based solely on the prior events ascribed to them in the trace, and the state of the network from all prior events. Thus, for a set of agents X with states $\mathit{State}_X = \{\mathit{State}_1, \ldots, \mathit{State}_{|X|}\}$, there is a transition relation

$$\mathit{Step}_{(P,\mathcal{A})} \subseteq (\mathit{State}_X, \mathit{Network}, \mathit{State}_{\mathcal{A}}) \times \mathit{Event} \times (\mathit{State}'_X, \mathit{Network}', \mathit{State}'_{\mathcal{A}}),$$

where transitions cause by events are restricted such that

1. if the event is performed by an agent x, then $(st, e, st') \in \mathit{Step}_x$ where st is the state of x in State_X. The new agent state State'_X is obtained by replacing the state of A in State_X with st', and $\mathit{State}'_{\mathcal{A}} = \mathit{State}_{\mathcal{A}}$,
2. if the event is $\texttt{recv}_x(m)$ for some agent x and message m, then $m \in \mathit{Network}$,

3. if the event is $\texttt{send}_x(m)$ for some agent x and message m, then $Network' = Network \cup \{m\}$ (note this is the only case in which an agent changes the state of the network),
4. if the event is performed by the adversary, then $(State_{\mathcal{A}}, e, State'_{\mathcal{A}}) \in Step_{\mathcal{A}}$, and $State'_X = State_X$.

We also require that events create deterministic a state transitions, i.e.

$$(st, e, st') \in Step_{(P,\mathcal{A})} \wedge (st, e, st'') \in Step_{(P,\mathcal{A})} \Rightarrow st' = st''. \tag{3.1}$$

The execution of a protocol is an alternating sequence of state tuples and events,

$$\langle st_0, e_1, st_1, \ldots, e_n, st_n \rangle$$

where st_0 is the initial state and each tuple $(st_{k-1}, e_k, st_k) \in Step_{(P,\mathcal{A})}$. The corresponding trace is the sequence of events $\langle e_1, \ldots, e_n \rangle$. We call the set of all possible traces arising from some protocol and adversary model a *trace set*, and use $Tr(P, \mathcal{A})$ to refer to the trace set of a particular protocol P and adversary $\mathcal{A}$. Note that trace sets are prefix-closed, as individual transitions are assumed to be atomic and the participants can stop at any time.

Generally we do not care about the specific events performed by the agents or the adversary, or their resulting encoding in the trace, other than requiring an abstract way to refer to certain events relevant to detection. This allows us to restrict the actions of participants as little as possible while still having well-defined communication structure. In addition to the special `send` and `recv` events which interact with the network, we name two other special types of events. We use $\texttt{compromise}(k)$ to refer to any adversary event that compromised some data k from any agent's state. This allows us to refer to, for example, the subset of traces in a trace set in which a particular term is never compromised. Finally, we denote detection of a compromised k by a special agent event $\texttt{detect}(k)$.

The initial state of the agents includes both agent-specific data as well as any public data assumed to be known both to the adversary and the agent (e.g. some settings may assume a public key infrastructure). The adversary's initial state contains only this public data. Since we do not bound the computation time of the agents or adversary, we instead assume a symbolic model of security in which a term algebra (e.g. that of TAMARIN [64]) defines how terms may be derived.

We focus on detection protocols that can automatically trigger an appropriate response when they detect, such as key revocation, disabling services, or blacklisting users. To enable this, it is important that there are no false positives. Formally,

Definition 4 (Soundly detecting protocol)**.** We say a protocol $P \in$ Protocol soundly detects misuse with respect to an adversary model $\mathcal{A} \in Adv$ if

$$\mathsf{sound}(P, \mathcal{A}) \equiv \forall tr \; . \; tr \in Tr(P, \mathcal{A}) \Rightarrow \Big(\forall k \; . \; \mathtt{detect}(k) \in tr \Rightarrow \mathtt{compromise}(k) \in tr\Big).$$

Note that completeness, in the sense of always detecting after compromise, is not possible in general unless compromise is directly observable by protocol participants. Nevertheless, it is possible to give guarantees of detection under particular conditions: for example, in Chapter 4 we will discuss protocols that guarantee detection in the session following particular adversary activity.

For shorthand, we enumerate some of the common sequence projections that we will use throughout this chapter to isolate particular parts of traces:

- For a set $X \subseteq$ Agent, $|_X$ for all trace events e such that one of the agents in X is performing e,
- $|_{c(k)}$ for all $\mathtt{compromise}(k)$ events,
- $\nmid_{c(k)}$ for all events that are not $\mathtt{compromise}(k)$ events,
- $|_{\mathtt{send}}$ or $|_{\mathtt{recv}}$ for all send or receive trace events respectively, and
- $|_{\mathtt{net}}$ for all network trace events (i.e. including both send or receive trace events).

In this thesis we do not prescribe any specific response mechanism for key compromise, since this is an orthogonal area of research (and often involves side-channels or other scenario- or system-specific resources). We instead discuss which parties can detect and when. Soundness enables any detecting party to immediately trigger whichever response mechanism it deems appropriate.

3.2.3 Reasoning about agents

In order to reason about agent capabilities, we must be able to talk about their state as well as the possible events they can perform under particular constraints. We begin with some notation to discuss the state of agents after a trace. Since trace events

are, by definition, enough to determine how agent state changes with each action, the state of some agents at some time along with a sequence s of events are sufficient to determine the state of those agents after s. This is formally stated in Corollary 6.

Definition 5 (State after a trace)**.** For a set of agents $X \subseteq \text{Agent}$, we introduce the notation $\mathsf{state}(tr, X)$ to represent the collective state of the agents of X after a trace tr.

Proposition 6 (State convergence)**.** Let T be a trace set and X a set of agents. Let $tr, tr' \in T$ be two traces such that $\mathsf{state}(tr, X) = \mathsf{state}(tr', X)$. Then

$$\forall s \; . \; tr \cdot s \in T \wedge tr' \cdot s \in T \Rightarrow \mathsf{state}(tr \cdot s, X) = \mathsf{state}(tr' \cdot s, X).$$

Recall from Equation 3.1, events fully determine each individual state transition. Thus, by definition the state of some particular agents X after a trace tr can be reconstructed entirely from the events in $tr|_X$. Therefore, it is necessarily true that $\mathsf{state}(tr, X) = \mathsf{state}(tr', X)$ if $tr|_X = tr'|_X$.

State convergence is a particularly useful property, because it implies that a subset of agents cannot differentiate two traces in which their combined states are the same, unless they later receive a message that is only possible in one of the two. In fact, we can lift this to prove practical limitations on when it is possible to detect even when agents can run an arbitrary protocol between themselves. We define *protocol extensions* to capture the events that could occur running a secondary protocol, without an adversary, after a particular trace.

Definition 7 (Protocol extension)**.** Let $T \subseteq Tr(P, \mathcal{A})$ for some protocol P and adversary $\mathcal{A}$. A protocol extension performed by a set $X \subseteq \text{Agent}$, beginning from a trace tr, is the set of all sequences of agent events s performed by agents in X such that $tr \cdot s \in T$, and s is independent of all prior network events. Formally,

$$PE(tr, T, X) \equiv \{s \;\mid\; (tr \cdot s) \in T \wedge (s|_X = s) \wedge \\ \forall m, i \; . \; \left(s_i = \mathtt{recv}(m) \Rightarrow \exists j < i \; . \; s_j = \mathtt{send}(m)\right)\}.$$

We use $PE(tr, T)$ as shorthand for $PE(tr, T, \text{Agent})$, which is equivalent to omitting only adversary events from the protocol extensions.

Intuitively, we will be using these protocol extensions to represent what a set of agents could determine by running a protocol amongst themselves *after* a particular

trace, in an ideal environment where no adversary interferes with them. This captures all the things the agents might collectively compute from their current state. State convergence can be leveraged to show a useful property of the protocol extensions across all possible protocols.

Lemma 8. *Let T be a prefix-closed set of traces such that $T \subseteq Tr(P, \mathcal{A})$ for some protocol P and adversary $\mathcal{A}$, and let $X \subseteq$ Agent be a set of agents. Then*

$$\forall tr, tr' \in T \ . \ (\textsf{state}(tr, X) = \textsf{state}(tr', X)) \Rightarrow PE(tr, T, X) = PE(tr', T, X).$$

That is, for every protocol, any two traces tr and tr' where $\textsf{state}(tr, X) = \textsf{state}(tr', X)$ have the same X-protocol extensions.

Proof. Assume otherwise; that is, without loss of generality there is a sequence s in $PE(tr, T, X)$ that is not in $PE(tr', T, X)$.

If $s \notin PE(tr', T, X)$, then by the definition of protocol extensions either $s|_X \neq s$, or $tr' \cdot s \notin T$, or there are `recv` events with no corresponding `send` in s. The first and last of these are trivially false by the requirement that $s \in PE(tr, T, X)$. Thus, it must be that $tr' \cdot s \notin T$; we will construct $tr' \cdot s$ recursively to show that this is false.

Take the first element of s, which we will call e such that $s = \langle e \rangle \cdot s'$ for some sequence s'. The set T is prefix-closed since it is generated by a protocol, and by the definition of protocol extension, $tr \cdot s \in T$, so $tr \cdot \langle e \rangle \in T$.

If e is a valid state transition after the state reached in tr but not after the state reached in tr', then from the definition of $Step_{(P, \mathcal{A})}$ it must be because either e is not a valid transition for the agent's state after tr' or e is an event $\texttt{recv}(m)$ for a message m that is not in the network after tr'. But both $\textsf{state}(tr, X) = \textsf{state}(tr', X)$, and the antecedent requires that all messages sent to the network are identical; in other words, the event e can be performed after tr', and thus $tr' \cdot \langle e \rangle \in T$.

By Proposition 6, $\textsf{state}(tr \cdot \langle e \rangle, X) = \textsf{state}(tr' \cdot \langle e \rangle, X)$. Since every trace is finite, the sequence s' must be shorter than s; so we recurse this argument over s'. If $s' = \langle\rangle$ then we are done, the trace suffix $\langle e \rangle$ was in both $PE(tr, T, X)$ and $PE(tr', T, X)$, a contradiction. If s' is not empty, then s' is a trace suffix in the set $PE(tr \cdot \langle e \rangle, T, X)$ but not $PE(tr' \cdot \langle e \rangle, T, X)$; we can repeat the argument above to remove the next event from the suffix and we find that every event of s is valid after tr'. □

Lemma 8 allows us to begin reasoning about the space of possible actions a set of agents can take. It shows that after a trace, a set of agents performing any protocol

at all amongst themselves are still limited to some computation over their collective state.

Note there is an equivalent definition of soundness in terms of protocol extensions.

Lemma 9 (Equivalent definition of soundly detecting)**.** *For a detection protocol* $P \in \text{Protocol}$ *and adversary model* $\mathcal{A} \in Adv$*,*

$$\mathsf{sound}(P, \mathcal{A}) \iff \forall tr, s \;.\; tr \in Tr(P, \mathcal{A}) \wedge s \in PE(tr, Tr(P, \mathcal{A})) \Rightarrow \\ \forall k \;.\; (\mathtt{detect}(k) \in s \Rightarrow \mathtt{compromise}(k) \in tr)\,.$$

Proof. Recall the definition of sound,

$$\mathsf{sound}(P, \mathcal{A}) \equiv \forall tr \;.\; tr \in Tr(P, \mathcal{A}) \Rightarrow \Big(\forall k \;.\; \mathtt{detect}(k) \in tr \Rightarrow \mathtt{compromise}(k) \in tr\Big).$$

If a protocol is sound by the definition above, then for all traces in $Tr(P, \mathcal{A})$, all $\mathtt{detect}(k)$ events must be preceded by a $\mathtt{compromise}(k)$ event. Since $tr \cdot s \in Tr(P, \mathcal{A})$ by the definition of a protocol extension, $\mathtt{detect}(k) \in s \Rightarrow \mathtt{compromise}(k) \in tr \cdot s$. Since $\mathtt{compromise}(k)$ cannot occur in the protocol extension s by definition, it must be that $\mathtt{compromise}(k) \in tr$.

In the other direction, let us assume that the implication is false: that `detect` events in protocol extensions imply a `compromise` event in the trace, but the protocol is not sound. If the protocol is not sound then, by the prefix-closed nature of trace sets, there must exist a trace $tr' \cdot \langle \mathtt{detect}(k) \rangle \in Tr(P, \mathcal{A})$ ending in a `detect` event, where $\mathtt{compromise}(k) \notin tr'$. Now consider the protocol extensions $PE(tr', Tr(P, \mathcal{A}))$. By definition, the event $\mathtt{detect}(k)$ must have been performed by one of the agents, and it is not a `recv` event; further, by our assumption, $tr' \cdot \langle \mathtt{detect}(k) \rangle \in Tr(P, \mathcal{A})$. But then $\langle \mathtt{detect}(k) \rangle$ must be a valid protocol extension of tr', and so tr' must contain $\mathtt{compromise}(k)$—a contradiction. □

3.3 Observation of misuse

Whether a usage of a key is 'correct' in general may not be possible to determine from the limited perspective of an agent. To detect misbehaviour, and subsequently attribute it to the misuse of a secret, the protocol (or in a wider sense, the security mechanism) must be designed to make the misuse observable by the detecting agent

in question. We first give two examples to provide intuition about the type of designs that fail to accomplish this, before providing a more formal treatment of observable misuse to build useful detection protocols.

Ideally, it would be possible to soundly detect any compromise by the adversary. There is however an upper bound on how much can be detected: intuitively, there is no possible protocol for a set of agents to soundly detect secret misuse if that misuse had no effect on them. We formalize this below, using the protocol extension properties discussed above.

Lemma 10 (Sound detection requires state)**.** *For a secret k, a set X of agents, and a trace tr in a prefix-closed trace set $T \subseteq Tr(P, \mathcal{A})$ generated by a protocol P with adversary $\mathcal{A}$,*

$$\forall s, tr' \,.\, s \in PE(tr, T, X) \wedge \mathtt{detect}(k) \in s \wedge tr' \in T \wedge \left(tr'|_{c(k)} = \langle\rangle\right) \wedge \\ \mathsf{state}(tr, X) = \mathsf{state}(tr', X) \Rightarrow \neg\mathsf{sound}(P, \mathcal{A}).$$

That is, if a set X of agents detect the misuse of k in a trace $tr \in Tr(P, \mathcal{A})$ when their state could also be reached in a trace tr' without compromise, then the protocol cannot be sound. This means that sound detection is necessarily impossible in a completely stateless protocol where $\forall tr \,.\, \mathsf{state}(tr, X) = \mathsf{state}(\langle\rangle, X)$, as well as after any situation in which the adversary can 'reset' an agent's state back to a state reachable in an uncompromised trace.

Proof. Assume it is possible for the agents in X to soundly detect after tr, and thus there exists a suffix $s \in PE(tr, T, X)$ where $\mathtt{detect}(k) \in s$.

The antecedent requires a trace tr' where

$$tr' \in T \wedge \left(tr'|_{c(k)} = \langle\rangle\right) \wedge (\mathsf{state}(tr, X) = \mathsf{state}(tr', X)),$$

and from Lemma 8,

$$PE(tr, T, X) \subseteq PE(tr', T, X);$$

thus $s \in PE(tr', T, X)$. Since the agents detect in s after a trace with no compromise events, the detection cannot be sound. □

The requirement that the state of the agents could arise in a restricted trace set (in this case, traces with no compromise of k) is a useful one, which we formalize in

terms of agent state being *consistent* with a trace set.

Definition 11 (State consistent with a trace set)**.** Let $T \subseteq Tr(P, \mathcal{A})$ for some protocol P and adversary $\mathcal{A}$. The state of some agents $X \subseteq$ Agent after a trace tr is consistent with the trace set T if there is at least one trace in T which leaves the agents in X in the same state as tr.

$$\mathsf{consistent}(X, tr, T) \equiv \exists tr' \in T \ . \ (\mathsf{state}(tr, X) = \mathsf{state}(tr', X)) \ .$$

We say that misuse of a secret k in a trace $tr \in Tr(P, \mathcal{A})$ is *unobservable* by a set X of agents when

$$\mathsf{consistent}(X, tr, \left\{t | t \in Tr(P, \mathcal{A}) \wedge \left(t \nmid_{c(k)} \in Tr(P, \mathcal{A})\right)\right\}).$$

Note that the set $\{t \ | \ t \in T \wedge t \nmid_{c(k)} \in T\}$ includes traces involving compromise of the key, so long as the compromise was not necessary for any events in the trace; this is to differentiate *compromise* from *misuse*. Unobservable key misuse necessarily limits the ability of the agents in X to detect; by Lemma 10 there is no idealized protocol the agents of X could run to detect the misuse. Doing so would also detect in the trace $tr \nmid_{c(k)}$, since by definition

$$\exists tr' \ . \ \mathsf{state}(tr, X) = \mathsf{state}(tr', X) = \mathsf{state}(tr' \nmid_{c(k)}, X).$$

It is important to note that, while observability of secret misuse is necessary for a set of agents to soundly detect it, it is not sufficient to guarantee that deciding whether to detect can be done tractably (i.e. in a polynomial amount of time). For example, consider a toy protocol where an agent generates a random value with some property and sends the output of a one-way permutation applied to that value over the network signed with their key—detecting misuse of that key may require inverting the permutation to check if the input value had the correct property.

3.3.1 Categorizing observable misuse

Lemma 10 shows that a set $X \subseteq$ Agent must reach a collective state inconsistent with the set of all traces without compromise of k to have a possibility of soundly detecting it. In this section we show a categorization of different ways an inconsistent state

might be reached, and prove some properties of them which should be considered when designing or modifying a protocol to detect secret misuse.

We divide the ways of observing misuse into three categories, based on the messages received by an agent. The first, *trace-independent inconsistency* refers to a received message that could not have occurred at all without compromise. The second, an *observation of contradiction*, refers to the observation of a sequence of messages which, while each individually possible to receive, could not be received in that sequence without compromise. Finally, an *observation of acausality* is when a sequence of received messages requires action on the part of an agent in order to occur in a trace set, but has occurred without such an action. This final type of observation requires agents to be in a position where they would know if the action did not occur.

Trace-independent inconsistency

The simplest way in which agents can determine that the current trace is inconsistent with a trace set is by receiving a message which could not occur in any trace of that trace set. This category of misuse event is observable 'statelessly' in the sense that it is inconsistent with the trace set independently of the current trace. As such, we refer to this category of observability as *trace-independent inconsistency*.

We formalize it as the negation of the predicate allowed, representing whether a message would occur in any trace within an arbitrary trace set T.

Definition 12 (Allowed messages)**.** A message m is allowed in a trace set T if there exists a trace in T containing a receive event of that message. Formally,

$$\mathsf{allowed}(m, T) \equiv \exists tr \in T \;.\; \texttt{recv}(m) \in tr.$$

Messages which are not allowed in a trace set are referred to as *disallowed* messages. For example, recall Example 1 from Section 3.1 in which a key is used to generate a message that would never occur when the parties involved follow the protocol.

This type of observability is relatively trivial. In practice one can rarely rely on the adversary sending a message which is, in itself, evidence of secret misuse. Nevertheless, it represents a special case of observable inconsistency with a trace set, in which no knowledge except the message itself is required to determine inconsistency.

Observing contradictions

If the messages in a trace are contradictory compared to a trace set that sequence of messages cannot occur in any trace of the trace set. This is formalized with the predicate contra.

Definition 13 (Contradictory messages)**.** Given a set $X \subseteq \text{Agent}$, a trace tr, and a trace set T, we say that the agents of X have received a contradictory sequence of messages when

$$\mathsf{contra}(X, tr, T) \equiv \forall tr' \in T \; . \; (tr|_X)|_{\texttt{recv}} \neq (tr'|_X)|_{\texttt{recv}}.$$

Note that receiving any disallowed message in a trace implies that the message sequence in that trace is contradictory, i.e. for any set $X \subseteq \text{Agent}$, trace tr, and trace set T,

$$\Big(\exists m \; . \; m \in (tr|_X)|_{\texttt{recv}} \wedge \neg\mathsf{allowed}(m, T)\Big) \Rightarrow \mathsf{contra}(X, tr, T).$$

Recall Example 2 in Section 3.1, in which two agents exchange messages with increasing counter values. An adversary taking the place of an agent temporarily can be detected in this case because it is possible to observe contradictory messages, as each message received from the other agent is expected to include an incremented counter value. An agent receiving two messages with the same counter value could observe that they contradict, even if each message would be allowed in the trace set where neither agent is compromised.

A stronger example making use of contradictory messages to detect is found in transparency overlays like Certificate Transparency, which we will discuss further in Section 4.1. Briefly, the ‘log’ in a transparency overlay signs particular messages for the protocol participants which are expected to be mutually consistent; misuse of the log’s key could therefore be detected by receiving two such signed messages that contradict each other.

Observing acausality

While the previous two categories reason about received messages, it is also possible to detect based on agent state directly by counterfactual reasoning. For example, an agent storing all prior uses of their key can identify misuse of their key if they receive a message which uses it that is not in their state. This extends in more complex ways:

the transparency overlays we will discuss in Section 4.1 are based on the ability of an identity or domain owner to determine when an entry in the log exists without some action on their part, on the assumption that only the owner should be initiating the process which leads to an entry in the log.

We define a notion of *violating causality*, where an agent can observe that the messages of a trace violate causality if they have some guarantee that they are required to participate in particular ways every time some sequence of messages occurs in the trace set.

Definition 14 (Violation of causality)**.** The messages of a trace tr in a trace set T are said to *violate causality* for a set of agents X if there is some trace in T in which those messages can be received, but no trace in T in which the same sequence of sent and received message occurs. Formally,

$$\mathsf{acausal}(X, tr, T) \equiv \Big(\exists tr' \in T \,.\, (tr|_X)|_{\texttt{recv}} = (tr'|_X)|_{\texttt{recv}}\Big) \\ \wedge \Big(\forall tr' \in T \,.\, (tr|_X)|_{\texttt{net}} \neq (tr'|_X)|_{\texttt{net}}\Big)$$

Since the agents of X would expect to have performed some particular actions before or during a sequence of messages, their state may become inconsistent with an uncompromised trace. In fact, the only way the agents' states can become inconsistent with a trace set upon receiving an otherwise valid series of messages is if they observe a violation of causality for that trace set. We formalize this in Lemma 15.

Example 3 in Section 3.1 can observe misuse because of the causal connection between messages. In the trace set where the adversary has not compromised either agent's key, the sequence of messages Alice receives from Bob are dependent on her sent messages, and thus a mismatched hash chain value would be evidence that their trace is inconsistent with that trace set.

Note, however, that in both the trace set where the adversary may compromise Alice's key, and the trace set where the adversary may compromise Bob's key, Alice cannot determine if Bob's hash chain value violates causality. In the former, the adversary can sign a different nonce value to Bob with Alice's key, and in the latter the adversary can sign a message as Bob directly with a different hash chain value. Thus, it is not possible for Alice to differentiate between these two trace sets upon receiving the wrong hash chain value: the trace is consistent with both.

Lemma 15 (Complete categorization)**.** *For a secret k, a set $X \subseteq$ Agent, and a trace set $T = Tr(P, \mathcal{A})$, let $tr \in T$ and $T_{uc} = \{t \mid t \in T \wedge t\nmid_{c(k)} \in T\}$. If tr leaves the agents of X in a state inconsistent with any uncompromised trace, then compared to the trace set T_{uc}:*

i) the message sequence observed in tr is contradictory, or

ii) the message sequence observed in tr violates causality.

Formally,

$$\neg\mathsf{consistent}(X, tr, T_{uc}) \Rightarrow \mathsf{contra}(X, tr, T_{uc}) \vee \mathsf{acausal}(X, tr, T_{uc}).$$

Proof. Assume this is not true, so that agent state after the trace is inconsistent with any uncompromised trace, but the trace does not contain contradictory messages nor does it violate causality. From this, we will reach a contradiction by constructing a trace in T_{uc} which leaves the agents of X in the same state as tr.

Note that T is generated by a protocol, so it is prefix-closed and thus by definition $\langle\rangle \in T_{uc}$. As such, tr must be non-empty, or it would be consistent with T_{uc}.

If the consequent is false, then expanding the definitions,

$$\left(\exists tr' \in T_{uc} \,.\, (tr|_X)|_{\texttt{recv}} = (tr'|_X)|_{\texttt{recv}}\right) \wedge$$
$$\left(\left(\forall tr' \in T_{uc} \,.\, (tr|_X)|_{\texttt{recv}} \neq (tr'|_X)|_{\texttt{recv}}\right) \vee \left(\exists tr' \in T_{uc} \,.\, (tr|_X)|_{\texttt{net}} = (tr'|_X)|_{\texttt{net}}\right)\right).$$

Thus, there exists some trace $tr' \in T_{uc}$ such that

$$\left((tr|_X)|_{\texttt{net}} = (tr'|_X)|_{\texttt{net}}\right).$$

If agent state after the trace tr differs from all traces in T_{uc}, then since trace events are by definition sufficient to construct the state of all agents and the adversary, there must be at least one event in tr performed by the agents of X which differs from their events in all traces of T_{uc}. Thus, there is some non-empty prefix $p \cdot \langle e\rangle$ of $tr|_X$ such that

$$\left(\exists tr' \in T_{uc} \,.\, tr'|_X = p\right) \wedge \neg\left(\exists tr' \in T_{uc} \,.\, tr'|_X = (p \cdot \langle e\rangle)\right),$$

where p may be the empty sequence and e is a single trace event. We separate the possible events for e into events that are not `recv` events, and those that are, and show that both lead to a contradiction.

Take any trace $tr' \in T_{uc}$ such that $tr'|_X = p$, which must exist from the above. If e is an event performed by an agent in X, but is not a `recv` event, then (by the assumption that agents can only interact through the network) it depends only on the local state of the agent. Since $\mathsf{state}(p, X) = \mathsf{state}(tr', X)$,

$$\exists tr \in T \; . \; tr|_X = (p \cdot \langle e \rangle) \Rightarrow (tr' \cdot \langle e \rangle) \in T.$$

But since $tr' \in T'$, and e cannot be a compromise event, then by definition

$$(tr' \cdot \langle e \rangle) \in T \Rightarrow (tr' \cdot \langle e \rangle) \in T_{uc},$$

a contradiction.

If e is instead a `recv` event performed by an agent in X, then it depends on both the state of the agents as well as the state of the network adversary. Since $p \cdot \langle e \rangle$ is a prefix of $tr|_X$, and there exists some trace in the prefix-closed set T_{uc} with all the same network events of tr by our original assumption, there must be some $tr_{uc} \in T_{uc}$ such that

$$((p \cdot \langle e \rangle)|_X)\,|_{\texttt{net}} = (tr_{uc}|_X)|_{\texttt{net}}.$$

In other words, it must be possible for the adversary to construct the message received in e in tr_{uc} without compromise.

Now, take any trace $tr' \in T_{uc}$ such that $tr'|_X = p$. Note that $(p|_X)|_{\texttt{net}} = (tr'|_X)|_{\texttt{net}}$ by definition. Since agents can only influence the adversary's state through `send` events, and those are identical for the agents of X in p and tr', the only way the receive event e might not be valid after tr' is through the activity of the other agents not in X.

Using this, we construct our contradiction. Consider a trace tr'_{uc} constructed from tr_{uc} by replacing the events by the agents in X with those in $tr'|_X$ in order, with the constraint that the `send` and `recv` events performed by X occur in the same places as before. This trace must exist in T_{uc}, as all events local to X are still valid regardless of the other agent activity, and all network events of X remain identical. But then, by construction, $tr'_{uc}|_X = (p \cdot \langle e \rangle)|_X$, a contradiction. $\square$

Note that in most cases, detection would only be feasible when the set of agents X that observes the misuse is a singleton. Nonetheless, knowing that some set of agents is able to observe misuse can be valuable for guiding protocol design, as it may be

possible to modify the protocol so that these agents can communicate enough to detect, or to narrow the number of agents required to observe misuse. Alternatively, for some systems it may be practical to assume some out-of-band channel for communication between the observing agents, and perform detection that way.

As an example, if a protocol requires at least one agent from a set to make a request before a particular token is produced, then that set of agents collectively have a causal role in the production of that token (and could therefore detect based on violations of that causality) but none of the agents individually do. However, if the protocol can be modified such that the token produced depends on *which* agent requested it, then each agent individually could have a causal role in the production of their own tokens. We will now distill lessons like these into general design principles and constructions.

3.4 Design implications and applications

We now revisit our results on categories of misuse observation to identify detection mechanisms and summarize them into some design principles for detection protocols. Finally, in Section 4.1 we re-examine transparency overlays with the additional context our foundations provide, and identify potential improvements.

3.4.1 Main detection mechanisms

In Section 3.3 we categorized three main mechanisms by which secret misuse can become observable. Ultimately, all of them rely on the observations that agents make through their interactions with the network. The difference in approaches mainly depends on the extent to which they take this information and their own actions into account.

Recall that state inconsistency is necessary but not sufficient for detection. Nonetheless, the categorization of observability conditions implies a categorization of the types of detection that can be designed into a detecting protocol, and some necessary conditions.

1. **Trace-independent observability** of any single message in the trace set, which requires no knowledge of the prior trace events.

2. **Contradicting observations** when a sequence of observed messages cannot

occur in a single honest trace. This requires enough knowledge of prior observed messages to determine if new observations contradict.

3. **Acausal observations** when the observed messages contradict the agents' knowledge of their own activity. This is only possible for agents who are in a position to observe violations of causality compared to honest traces, and it requires enough knowledge of past agent actions as well as prior observed messages to determine whether the agent caused the observed messages.

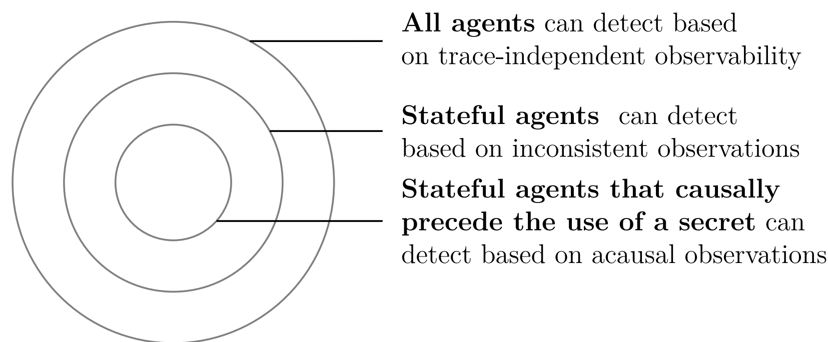

Figure 3.3: Venn Diagram of the type of agents and the detection mechanisms that they might be able to use. Only stateful agents that have a causal role the use of a secret could make use of all three types.

3.4.2 Design principles

The combination of the three types of detection leads to a number of design principles for detecting the misuse of secrets.

We note that for any given application, there may be practical and security considerations that affect whether and how the principles can be applied. For example, the wish to maintain confidentiality and unlinkability of messages may limit the application of Principle 3. Restrictions on message size, communication complexity, and storage size may limit the applicability of any of the principles. This directly results in a trade-off between such restrictions and the ability to detect the misuse of secrets.

- **Principle 1**: Protocol messages should be tightly coupled to prior messages. This helps maximize the possibility of any misuse detection, and prevents an adversary from 'resychronizing' agents after misusing keys (e.g. the attack described in Example 2). Stateless protocols necessarily violate this principle.

- **Principle 2**: Include unique and unpredictable values in messages. This helps to establishing contradicting observations, and ensure an adversary cannot correctly predict what an agent will do next. If values are not unique, then agents could get identical observations from messages sent at different points, making them indistinguishable. If an adversary could predict the next exchange, they could potentially carry it out in advance with one of the participants and then take their place in the real exchange without leaving any evidence.

- **Principle 3**: Maximize the spread of data that other parties might find contradictory or acausal. Detection requires observations, so it is important to increase the opportunities for that to happen. Ideally, some observations could be broadcast to all participants (e.g., used when disseminating transactions in Bitcoin-like systems [43, 52, 69] to detect double spending), but for many applications this is not feasible. This motivates the need for compromise solutions such as a gossip protocol (e.g., [20]).

- **Principle 4**: Identify which agents can observe violation of causality for important messages, and ensure they can observe those messages. Agents who are required to trigger a sequence of messages can detect more than agents who can only detect by observing contradictions. It is therefore worthwhile to ensure that the protocol enables the detection of acausal observations as much as possible.

  For example, in the PKI setting, the agents which can observe violations of causality are the domain owner and the CA, since a certificate for a domain signed with a CA's key should only exist after it has been requested by the domain and then signed by that CA. If such a certificate occurs without the request, or without the CA signing it, then the key must have been misused. This principle is implicitly used in systems like Certificate Transparency [59] and other systems based on transparency overlays, which we will return to in Section 4.1.

Some minor aspects of the above principles are similar to principles from earlier work [2], but there are crucial differences. Principle 1 explicitly requires state, which leads to a trade-off between security guarantees and keeping track of state. Principle 2's unique values have been suggested before, but not all messages need to have unpredictable values for other security properties. This unpredictability is specifically useful for

detection. Principle 3, which suggests spreading data, improves detectability at a clear cost in terms of transmissions, which would be avoided by previously proposed principles (except perhaps accountability). To the best of our knowledge, Principle 4 is entirely derived from our detection-based observations, though it is implicitly used in some systems.

Next, we will apply the intuition and principles built in this Chapter to reconsider a class of existing designs for detection, called transparency overlays, and develop novel protocols to detect secret misuse.

Chapter 4

Applications of detection theory

With the the foundations established in Chapter 3, we now have the intuition necessary to analyze existing protocol designs and instantiate new protocols with detection properties. In this chapter, we will begin in Section 4.1 by examining transparency overlays, and propose a concrete improvement based on our work. We then present two types of concrete protocols which can detect misuse—one based on counters, and the other based on so-called commitments. We consider these example protocols as a template for modifying existing protocols, and demonstrate this for each protocol by examining a scenario where secret misuse is a concern and modifying an existing protocol to achieve detection properties.

In Section 4.2 we present a type of detecting protocol based on counters. We present an example protocol which has minimal state and communication requirements, while still providing a detection guarantee when an adversary is only capable of compromising a subset of keys. This is often the case in real systems, and we take a recent development in Content Delivery Networks (CDNs), a protocol called Keyless SSL [22], as a case study. We discuss Keyless SSL in more detail in Section 4.2.2, but in short Keyless SSL is designed to allow a customer of a CDN provider to maintain control over the private keys for their web certificates, by having the customer provide a signing oracle for the key. We show that by embedding our counter-based example inside the key exchange used by Keyless SSL, it is possible for the customer to detect some misuse of—and automatically revoke access by—a CDN server's key, with minimal overhead.

Finally, we describe a type of detecting protocol based on providing commitments to the contents of other sessions. We give an example protocol which provides detection guarantees even when both parties involved may be compromised, and can detect an

adversary attempting to authenticate as an agent unless they have compromised the full state of that agent since their last session. We consider this example in the context of systems that require high assurance, and in particular the Common Access Card (CAC) [91], the standard identification for United States Armed Forces personnel. Our commitment-based detection protocol can detect and revoke a cloned card on the next use of the original, while also preventing the stockpiling of cloned cards for use at a later time. We demonstrate this by modifying a standard authentication protocol used by smart cards, ISO-IEC 9798-3-3 [40].

After our work improving verification of stateful protocols in Chapter 6, we will return to the counter and commitment protocols in Chapter 7 in order to formally analyze their security properties.

4.1 Improving transparency overlays

Transparency overlays and related public log-based systems [9, 10, 19, 59, 98, 99] are designed to make participants' behaviour public through the use of a third-party log, enabling misuse detection on the basis of acausal observations. To avoid having to trust the log maintainer, transparency overlays make use of a log structure where the maintainer must be able to prove that any two log states they authenticate are consistent with each other. This allows misbehaviour by the log to be detected through observation of contradictory log states, and furthermore this misuse can be proven to other participants. In fact, this is precisely our **Principle 3**—to distribute information as widely as possible—and is one of the core developments underlying transparency overlays.

With an authenticated log that allows misuse of the log's key to be detected, it becomes possible to build a system in which acausal actions by another party can be observed. Participants making use of a transparency overlay can examine log entries to ensure both that every relevant action they see has an entry in the log, and that all entries in the log appear correct. This allows participants to observe acausal entries in the log, even where they would normally not be a part of a session making use of it. For example, detection of a mis-issued certificate in Certificate Transparency may be done by domain owners checking the log and discovering a log entry for a certificate that they did not request, as mentioned in **Principle 4** above.

Transparency overlays thus have two interlinking parts which make use of detection:

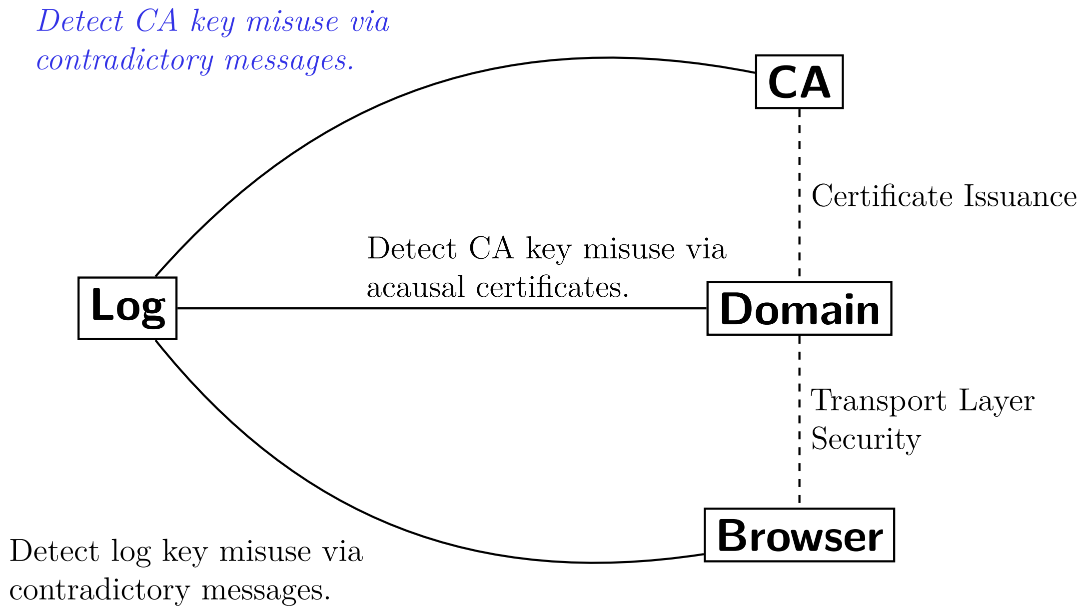

Figure 4.1: Certificate Transparency's interactions with existing web PKI (shown dotted lines), with the types of detection employed. Our proposed modification is shown in blue italics.

the first comparing the ongoing authenticated log states to ensure they do not contradict each other (which can be done by anyone), and the second auditing the contents of the log for entries indicating acausal use of a secret (which can be done only by particular parties for each entry). In Certificate Transparency, these are typically performed by individual users' browsers while navigating the web, and by domain owners respectively, as shown in Figure 4.1.

Recall from our foundations above that observing contradictory messages requires less than observing acausality. Since we can detect acausal uses of the CA's key, what about contradictory uses? Surely if we can perform the former then there must be some modification which allows for the latter (though it might be impractical to implement).

Based on our design **Principles 1 and 2**, we propose that CT-like transparency applications can be extended to allow dependencies between submissions from the same source, adding a further line of defense to transparency overlays and improving attribution when misuse is detected. Taking Certificate Transparency as a canonical

example, we propose to add into each log submission a value dependent on the previously submitted certificate—for example, the hash of the previous submission—so that the log can perform a first-line check for misuse. This modification is shown highlighted in Figure 4.1.

With an addition that allows log commitments to contradict each other, a log server can determine whether the CA knew about the previous commitment authenticated by them, or whether it might have been committed without their knowledge. If the certificates do not contradict each other, then this is evidence that the certificate authority's state has been updated with each certificate issued. On the other hand, if they do contradict each other, this indicates that either the CA's key has been misused by an adversary, or the CA systems are failing to correctly track each certificate they issue.

When contradictory certificates are submitted to a log server, the log server can swiftly notify the CA that either its key has been compromised or its system has not been updated with all issued certificates. On the other hand, if all certificates in the log are consistent with each other, then a domain owner discovering a mis-issued certificate for its domain in the log knows that the CA's own system must have been updating their state when the certificate was issued—an indication that the CA should have some record of issuing that certificate.

Implementation considerations Our proposed additions (as applied to CT) make the CAs stateful in their creation of certificates, though with negligible overhead introduced; arguably, CAs are already expected to keep an internal audit trail for each certificate issued.

Importantly, the state kept by the CA depends only on local operations, and not on any feedback from log servers. No latency is introduced into the process of issuing certificates, a facet that is vital for the modification to be practical. If all new certificate submissions to the log in Certificate Transparency were required to include this information, it would immediately benefit detection of CA key compromise.

This proposal is only one example of an addition which would force contradictions between log submissions from the same submitter in a transparency overlay. More elaborate constructions like the consistency proofs used by log servers could be leveraged to make submissions to a log from a misused key contradict a larger set of prior entries, for additional redundancy or for tying together multiple independent logs. In other transparency overlay applications, the commitment protocol shown in

Section 4.3 could be used to ensure that future log submissions come from the same party that generated the pre-commitment in the prior entry.

In practice, implementing this change in systems using transparency overlays would mean that misuse of a key to submit to a log could be detected rapidly if there are still ongoing honest submissions. Furthermore, it would give some assurance that all log entries were at some point known to the submitter, since they must have updated some part of their state with each previous submission. Though this modification does not replace the need for detection of acausal activity in the log, the benefits it provides can be done with negligable overhead on the part of both the log submitters and the log itself. The change would enhance the ability of transparency overlays to provide detection guarantees, and narrow down the potential issues to investigate if misuse is detected.

4.2 Counter-based detection

A simple approach to ensuring a causal relationship between messages in an authentication protocol is to have parties count the number of successful authentications. If an agent has some causal role in counter increases when the key is uncompromised, then when they observe violations of causality they can determine that a key was misused.

This simple approach of counting authentications has been applied in other domains in the past. An early example can be found in the late 17th century, with a lock designed by the locksmith John Wilkes [94] to count the number of times it has been opened. This allows an owner who remembers the value of the counter each time they open it to be sure that only they have opened the lock. More recently, a similar construction of a physical lock was patented in 1974 [15], likewise with the goal of allowing the owner to detect when other parties successfully 'authenticated' to the lock.

In protocols, an early, flawed, attempt at using counters to provide clone detection was in the ANSI-41 specification for cell phone networks [1] that we discussed in Section 2.3.1. The failure of this protocol to provide useful guarantees highlights the importance of both understanding and formally verifying the logic of detection: while the locks above only increment their counter when they are unlocked, the mechanism in ANSI-41 did not, instead incrementing the phone's local counter each time it *attempted* to authenticate. In our terms, this means that there was no specific cause of

the counter increments (at least, from the network provider's perspective); this limits their ability to detect to, at best, the observation of inconsistent counter values. Since most counter values were never received in an unreliable network and the counter frequently rolled over back to the initial value, the mechanism rarely worked for this purpose.

Here, we present a simple counter-based detection protocol based on asymmetric operations, and discuss verification of the protocol's properties in Tamarin. There are several scenarios where this protocol can serve as a blueprint for modifying existing protocols to provide detection. As a case study, we take an example relevant to internet infrastructure: the Keyless SSL protocol introduced by CloudFlare [22]. We describe the Keyless SSL protocol in Section 4.2.2 and present a modified handshake that provides detection guarantees.

4.2.1 A counter-based detection protocol

The *counter-based* detection protocol shown in Figure 4.2 is based on the notion of causality violation discussed in Section 3.3.1. The intuition behind the protocol is to ensure that counter increments are caused by recent uses of a signing key. The initiating agent I increments their counter only when they receive a freshly-signed message (i.e., including a nonce generated by I) from the responder R, and R only updates their stored counter value when they receive a similarly freshly-signed message from I to complete the session. With this, it becomes possible for I to observe acausal messages, either due to R's counter incrementing without I having sent a corresponding signed message, or due to I updating the stored counter value without a corresponding increment signed by R.

The counter-based protocol shown also provides a useful illustration of how our foundations clarify the limitations of a protocol. In the trace set where neither $\mathsf{sk}(I)$ or $\mathsf{sk}(R)$ are compromised, the responder R can determine if either of the messages m_1 or m_2 violate causality, which includes the incremented counter value to be checked, and the initiator I can do so for m_2 before completing the session (and thus before beginning the next session with an incremented counter). Because of this, it is possible for I to soundly determine that a key must have been misused if they observe a mismatch between their stored counter, and a counter increment signed by R.

However, m_2 containing an incremented counter without action by I does *not* violate causality in the trace set where $\mathsf{sk}(I)$ may be compromised (even if $\mathsf{sk}(R)$

cannot be), because the adversary can take on I's role in the protocol. Further, it would not violate causality in the trace set where $\mathsf{sk}(R)$ may be compromised (even if $\mathsf{sk}(I)$ cannot be) as the adversary can simply generate m_2 on their own without action by I. Therefore, it is impossible for I to determine *which* of the two keys were compromised from observing an acausal counter value.

From our foundations, it also becomes clear how the protocol could be modified in order to allow this: I must be required to send messages for all responses from R in m_2, even when $\mathsf{sk}(R)$ is compromised. This could be done, for example, by having R also include the signed message it received from I in the prior session. An adversary misusing $\mathsf{sk}(I)$ in this modified protocol would lead to the real I being presented with an incorrect counter value along with a message signed by $\mathsf{sk}(I)$ that I never sent. Misusing $\mathsf{sk}(R)$, on the other hand, would lead to an invalid counter update combined with a signed message that was sent by I.

We present here the simpler version of the protocol, as the proposal above still fails to detect when both keys may be compromised. In many practical situations, such as the Keyless SSL case we examine later, only one key is of specific concern, and in Section 4.3.1 we present a protocol which can detect misuse even when both keys are compromised.

The counter protocol has the following detection properties. In Section 7.1.1 we will revisit this protocol to model and analyze it, and formal statements of these properties in our semantics can be found there.

Sound detection. If there is a `detect` event triggered for a particular key pairing, then at least one of the keys in that pairing was compromised before the `detect` event.

Guaranteed detection of misuse of the initiator's key in prior sessions. If only the initiator's key can be compromised, then any session violating agreement on session data will be detected during the next session in which the adversary does not interfere.

4.2.2 Keyless SSL

Content delivery networks (CDNs) are services which provide many geographically dispersed caching proxies for internet content. CDNs are designed to increase availability and performance of internet services, by providing redundancy of content

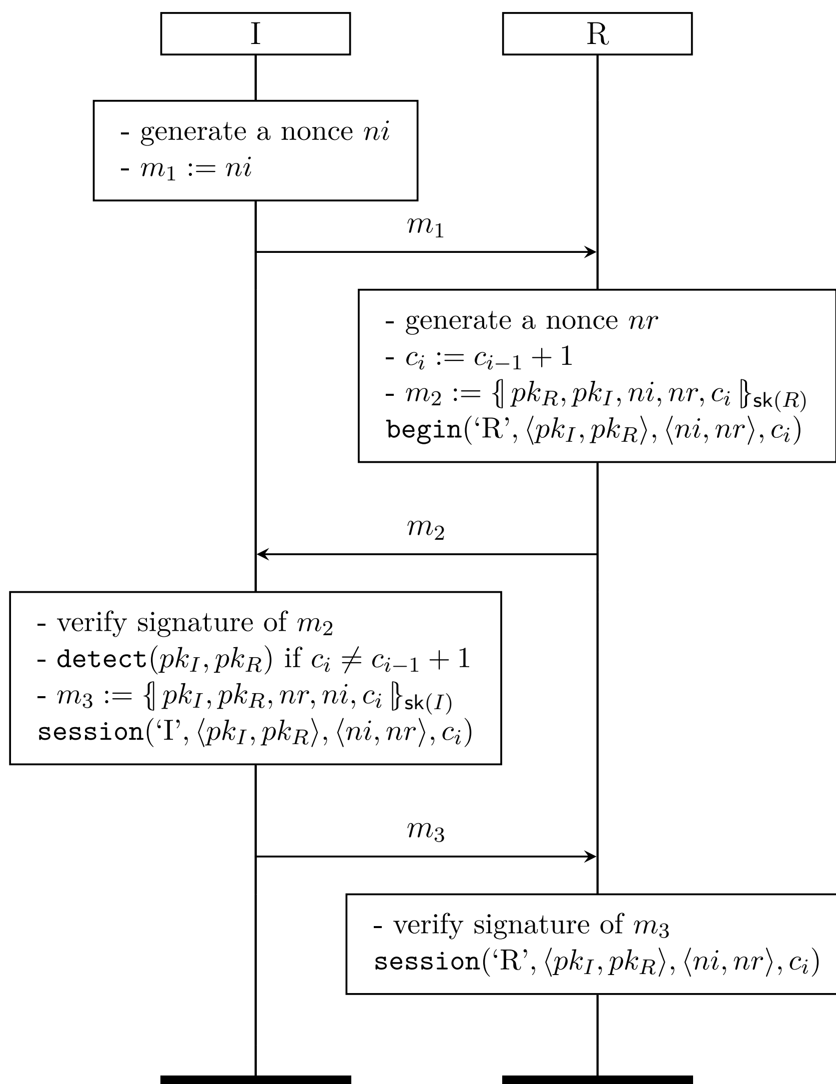

Figure 4.2: The i-th session of the counter-based detection protocol. The initial message with ni can be removed at the cost of additional state, if the initiator instead generates the nonce for message m_2, and the responder begins with m_1 by including the ni from the previous session.

hosting and lower latency to end-users (because of geographic proximity), and they can do so mostly transparently, even as a third-party [32]. This has made CDNs extremely popular, either as a service that can be purchased from CDN providers or as company-specific infrastructure for many internet companies; CDNs carried over half of all internet traffic in 2016 [88].

Purchasing CDN services from a third party, however, presents an obstacle to the use of TLS. Since the CDN's server comes between the end-user and the origin of any requested content, the CDN must be able to terminate TLS connections as if it were the origin. Naïvely, this would mean that each of the CDN's servers must have a private key corresponding to a certificate for the origin domain—a worrying proposal for some of their customers, and potentially even illegal in some jurisdictions and sectors. To alleviate these concerns, Keyless SSL was developed.

Keyless SSL is a protocol designed by a CDN provider, CloudFlare, to allow the provision of Content Delivery Network (CDN) services to their customers which have domains that do not want to, or cannot cede the private keys associated with their certificates to CloudFlare [22]. In Keyless SSL, CloudFlare's servers interact with a key server provided by their customer (e.g. a web service), using it as an oracle to complete a key exchange on that web service's behalf. This allows CloudFlare to carry out TLS handshakes as if they knew the web service's private key, while it remains secure on the web service's hardware.

In practice, this means that a large number of different private keys are each sufficient to use that web service's key server as an oracle, though with much greater control over key issuance and revocation than in a typical TLS environment. This makes detection of key misuse especially valuable—key revocation and remediation are much easier than they would be for a compromised domain certificate. By modifying the Keyless SSL protocol to include the same concepts as the counter-based example above, we can ensure that events from the web service must send a message to cause counter increments, so that they can detect if the CDN server's private key has been misused in any prior session. Since a CDN server carries out a new TLS handshake with the web service's server frequently (roughly every two hours), any misuse of the private key can be swiftly detected, and the key can be revoked by the web service before completing the session.

As in the example counter-based protocol above, the detection guarantee in our modified protocol requires that only one participant is compromised. Specifically,

we assume that the oracle's key is uncompromised; this is justified by their role as a signing oracle in the Keyless SSL protocol. If both the CDN and the web service can be compromised, then it is possible for the adversary to avoid detection by re-synchronizing the CDN server's counter, running sessions as the web service But in this case, the adversary could also use the web service's key directly to impersonate them without involving the CDN.

The implementation of the modified protocol presented here would allow the web services hosted by the CDN to have assurance that their key server has either not been accessed by an adversary who has gained access to any of the CDN server keys, or if it is being accessed by the adversary then it will be detected in short order. Furthermore, the web service can revoke a detected key immediately and automatically before completing the key exchange, preventing further damage from the compromised key. The proposed protocol requires very little modification and minimal storage requirements: a single counter value for each CDN server.

We briefly describe the protocol flow shown in Figure 4.3 before discussing its properties below. We consider the mutually authenticated TLS handshake performing a Diffie-Hellman exchange (DHE) between a CDN server C (the initiator), and one of their customers, a web service owner W[1]. C and W hold secret keys $\mathsf{sk}(C)$ and $\mathsf{sk}(W)$, respectively. They also have some means to validate each other's public keys–typically, $\mathsf{pk}(W)$ would be provided through some authenticated out-of-band communication while $\mathsf{pk}(C)$ is signed by a CDN-specific CA known to W. In this setting, we wish to provide some security guarantee against an attacker who obtains $\mathsf{sk}(C)$ and all state information (i.e. nonces) of C generated in any session. The goal of our protocol modification is to detect the compromise of $\mathsf{sk}(C)$.

In the first session, the counter begins at some known value, say '0'. In the i-th session, when C is establishing a shared secret with W, C begins a mutually-authenticated TLS exchange by creating a new nonce nc_i and sending the first message (known as the `ClientHello` message in the TLS protocol) to W. Upon receiving this message, W generates its own nonce nw and replies to C with, among many other things, a signature on nc and nw in the second message. Note that the exchange so far is unmodified from the standard TLS mutually-authenticated DHE.

Upon verifying the signature in the second message, C is certain that $\mathsf{sk}(W)$ is

[1]Keyless SSL allows only two cipher suites in the TLS handshake with the oracle, both of which use an elliptic curve Diffie-Hellman exchange [22], so we do not need to consider modifications or analysis of other modes.

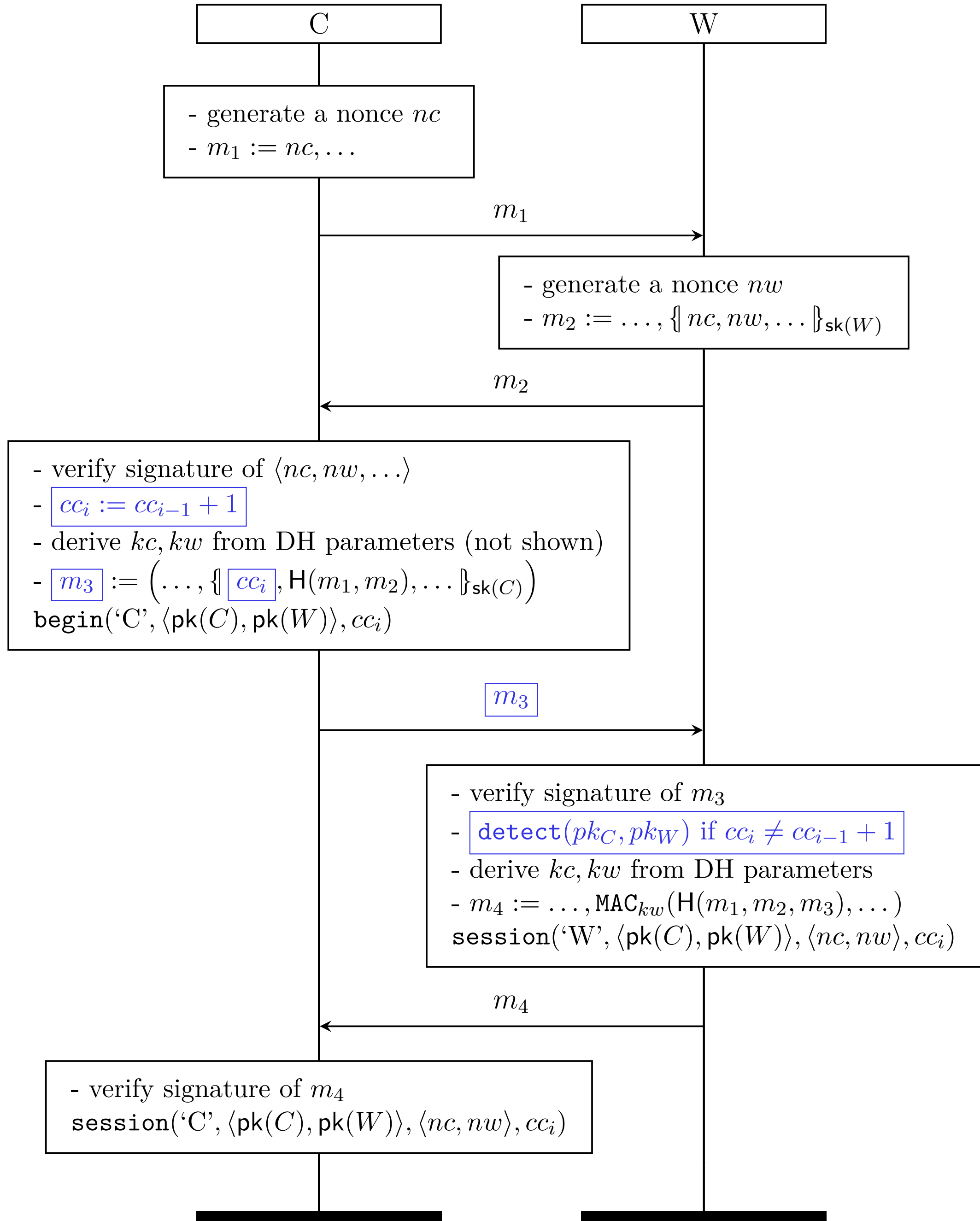

Figure 4.3: An example of the additions to the i-th session of the TLS mutually-authenticated key exchange used in Keyless SSL. For clarity, we omit terms in the messages that are not relevant. The modifications for detecting misuse are boxed and highlighted in blue.

being actively used in the current session, and so increments its counter cc_i. This counter value is then included in m_3. This is the only modified message of the protocol.

If cc_i does not match what W expects but the hash and signature are valid, a detection event will be raised and W can revoke C's key immediately to limit potential damage. W can later contact C through an out-of-band channel to begin remediation and attempt to discover the source of the compromise.

The modified Keyless SSL protocol exhibits similar properties to the counter example above, and we will formalize these properties and discuss the formal analysis of this protocol in Section 7.1.2.

Sound detection. If there is a `detect` event triggered for a particular key pairing, then at least one of the keys in that pairing was compromised before the `detect` event.

Guaranteed detection of misuse of the CDN key in prior sessions. If only the CDN's key can be compromised, then any session violating agreement on session data will be detected during the next session in which the adversary does not interfere.

4.3 Commitment-based detection

A more complex approach to a detection protocol is one applying ideas from key rotation schemes (see Section 2.2.3) in ways to facilitate detection as well as prevent adversary action. In this section we present a protocol based off an agent generating a new key each session, and providing a 'commitment' to that secret. In the following session, the agent must provide proof that they still have knowledge of the secret in order to authenticate. If an adversary has not recently compromised the agent, then they will not know the secret necessary to fully authenticate; if they have, then the next honest session will be using the previous secret, allowing for detection of misuse.

The technique presented in the commitment-based protocol below is ideal for situations where a high degree of security is required. We consider one particular scenario as a case study, the use case of the Common Access Card. The Common Access Card (CAC) is the standard identification card for United States Defense personnel, and has been used as an authentication token for security network systems and also for physical access to sensitive areas [16]. It supports asymmetric key cryptography and has writable memory. The CAC provides a useful example of a high-security domain where it is valuable both to detect if a cloned card has previously been used,

as well as 'heal' compromise so that any clone becomes useless unless immediately used.

As the protocols used by the CAC are not public, we show the applicability of our commitment-based protocol to this domain by modifying and verifying a protocol used in similar cards, the ISO-IEC 9798-3-3 authentication protocol [40]. In this instance, the initiator I in the 9798-3-3 protocol is a card reader connected to a back-end server, and the responder R is the CAC.

4.3.1 A commitment-based protocol

The design implications in Chapter 3 suggest that the message sequences in a protocol should be tightly coupled. Taking this to an extreme, in the commitment-based protocol every session includes not only a term established in the previous session, but also a term which commits the agent to some aspect of the *next* session. Naïvely, this might be done by generating a random value and providing the output of a one-way function in one session, followed by presenting the original value in the following session. But while this may be sufficient when there is no adversary on the network, in a Dolev-Yao setting the adversary could make use of the revealed value to insert their own session. Instead, we prevent this with a construction which allows an agent to commit to some property of the next session for which the corresponding proof is both coupled to the session data, and does not reveal the means to construct a different proof.

We show an example of such a construction with the commitment-based detection protocol shown in Figure 4.4. In this protocol, R generates an asymmetric key pair and presents I with a fresh commitment constructed by signing session data with the secret key, as well as the secret key used for the previous commitment to ensure continuity. Note that at the time the commitment is made, I is not capable of validating it directly, beyond knowing that it's associated with the previous commitment through the signature under the proof key. In the session following, R provides the public key that allows I to verify that the commitment is correct based on previous session data.

At no point does R reveal the key used to construct the commitment, ensuring that the adversary cannot authenticate their own session data even if they trick R into revealing an arbitrary number of commitments and their proofs. Instead, the proof is provided for the previous commitment while 'liveness' of the corresponding secret is proven in the next session, by using it to sign the next commitment.

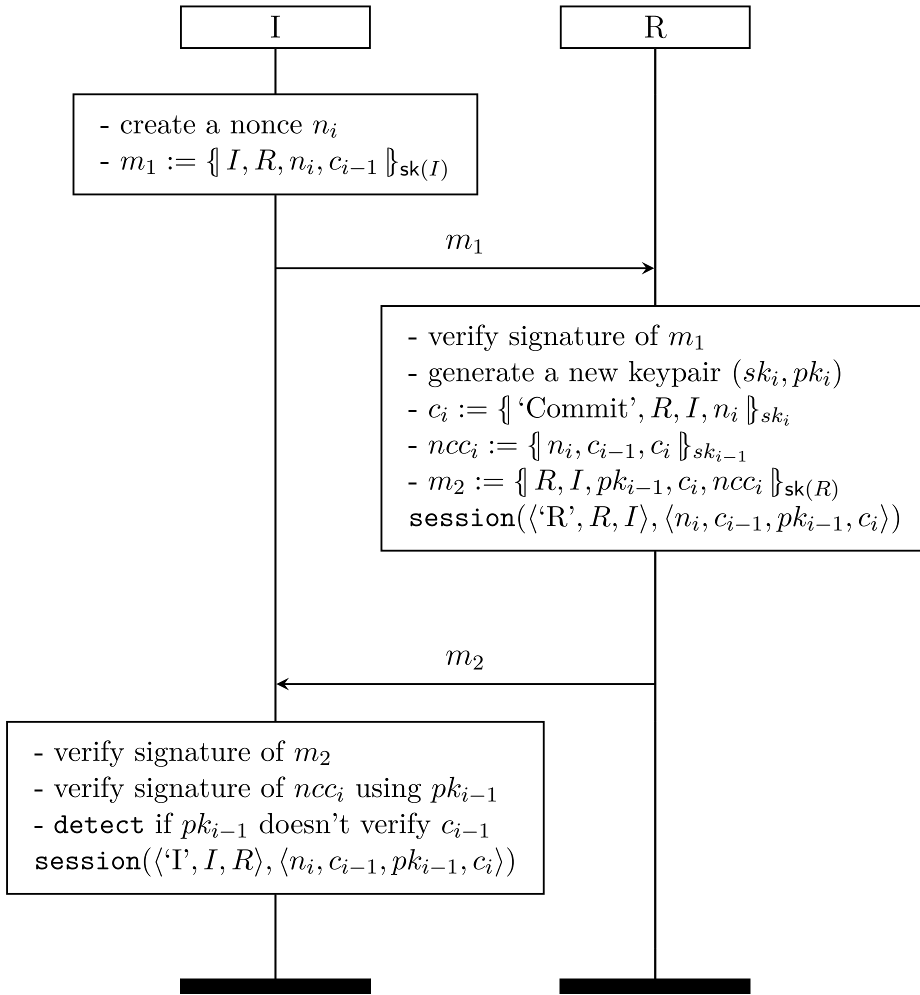

Figure 4.4: The i-th session of our example commitment-based detection protocol.

The commitment protocol has a number of desirable detection properties, which we will verify in Section 7.1.3.

Sound detection. If there is a `detect` event triggered for a particular key pairing, then the responder's key was compromised before the `detect` event.

Guaranteed detection of misuse in prior sessions. Any session violating agreement will be detected during the next session with agreement, even if the adversary is allowed to compromise all state and any one of the keys.

Guaranteed detection of misuse in future sessions. If there was a previous session with agreement and the adversary has not revealed R's state since that session, then any session violating agreement will be detected, even if the adversary has compromised all keys.

4.3.2 Modifying the ISO-IEC 9798-3-3 protocol

The ISO-IEC 9798 standard defines a family of entity authentication protocols, ranging from one- to five-pass symmetric and asymmetric key based authentication protocols. These protocols are not numbered directly, but Basin *et al.* provide a useful nomenclature in [8] from which we refer specifically to the 9798-3-3 protocol. The 9798-3-3 protocol is a two-pass mutual authentication protocol using digital signatures, and its simplicity makes it straightforward to use in settings where secrecy is not required and some kind of public key infrastructure exists, as in the case of smart card authentication. As such, we use it as an example of a generic smart card authentication protocol which may be used to authenticate to a wide range of different services, as with the Common Access Card. Note that in [8], Basin *et al.* demonstrate that the 9798-3-3 protocol is vulnerable to a reflection attack if an agent may take on both roles; in the following we assume that the reader and card (the initiator and responder respectively) are always assigned different signing keys, preventing this attack. This is typical in real deployments, where the key used by the card is generated on the card or during manufacturing, to prevent keys from being exfiltrated from the card.

Smart card authentication is used in a number of domains, some with millions of users each with their own unique card. These cards are used to authenticate access to, for example, buildings, credit accounts, networks, or online services. Card cloning

is an occasional concern, either because of poor cryptography [53], or various side channel attacks allowing extraction of keys, like differential power analysis [72, 92] and—at much greater expense—electron microscopy combined with chip probing [47]. Most smart card security is built around a substantially increased cost for an attacker to clone a card, and infeasibility of cloning on a large scale, rather than an assumption that it can protect against a targeted attack with physical access to the card.

In the United States, the Common Access Card is used for authentication in all of the domains listed above, and is the standard identification for all active-duty defence personnel. The security-critical nature of military resources makes targeted cloning attacks a more direct threat. The scale of the system even makes attacks by card suppliers a concern, as a supplier could build in a method to ex-filtrate keys or stockpile them at the time of manufacture. Rapid detection of cloned cards, even at the time they are first used, would provide additional assurance on top of existing technology.

In Figure 4.5, we present a modified 9798-3-3 protocol that provides strong detection properties. We show that it is possible for a smart card authentication protocol to not only swiftly detect and revoke cloned cards, but also invalidate any previously existing clones every time a card authenticates. This is done in such a way that an attempt to use an earlier clone results in immediate detection and revocation of privilege *before* the card successfully authenticates. Furthermore, this is possible even if the adversary can also compromise the key of the reader, *and* intercept messages between the card and the reader. We briefly describe the protocol flow here before formalizing these properties in the following section.

At the initialization phase, each CAC stores a unique commitment c_0 created by the back-end server. In the i-th session, when a reader detects a CAC, the reader communicates with the back-end server, which generates a message m_1 signed with the server's private key $\mathsf{sk}(I)$ containing the identities of the server and card, a fresh nonce ni_i, and the previous commitment provided by the card c_{i-1}. The reader forwards this to the card.

The card verifies the signature, and that the provided c_{i-1} agrees with its local memory. If these verifications succeed, the CAC generates a new nonce nr_i and a pair (sk_i, pk_i) of signing and verification keys. The CAC creates a new commitment c_i using sk_i, signs c_i again using sk_{i-1}, and uses its long-term key $\mathsf{sk}(R)$ to create a signed message m_2 from the identities (R, I), the two nonces (nr_i, ni_i), the signed

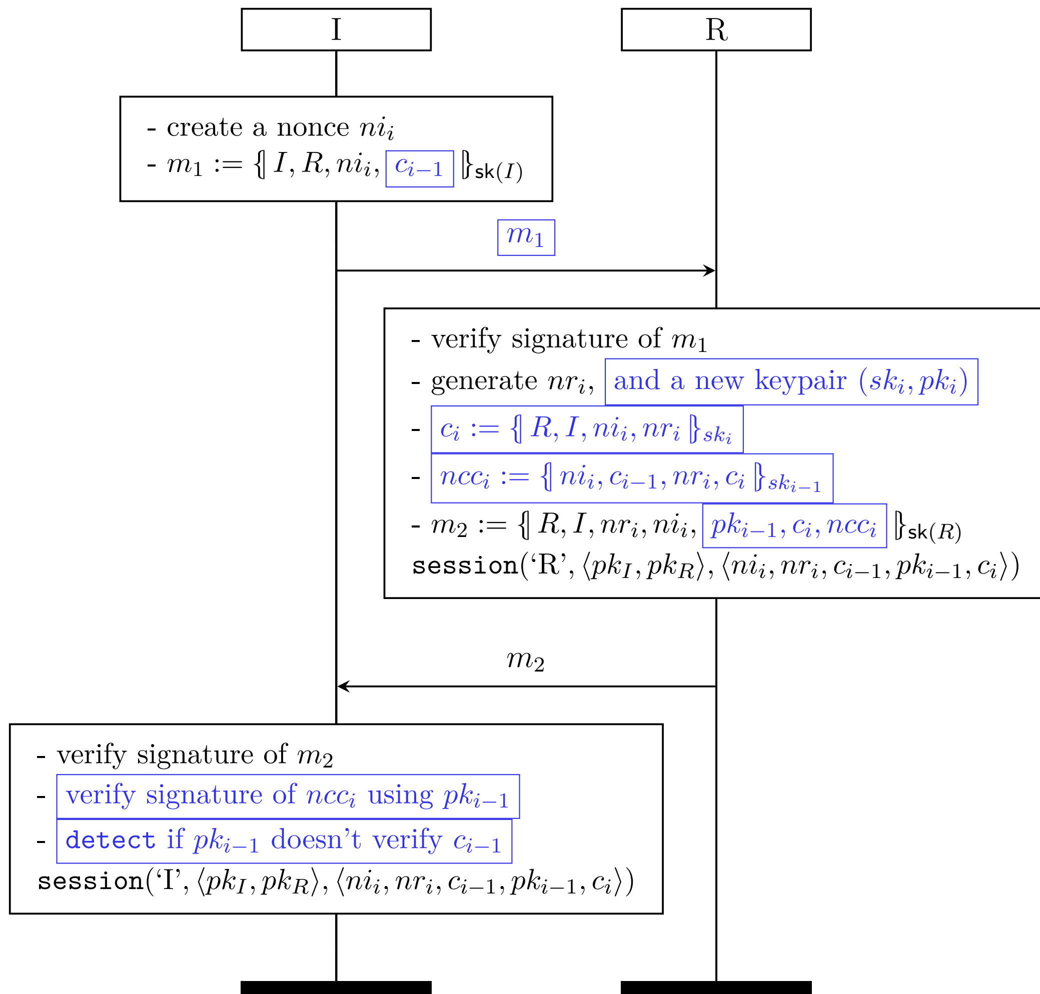

Figure 4.5: The i-th session of the modified ISO-IEC 9798-3-3 standard protocol. Modifications are boxed and highlighted in blue.

c_i, and the public key that verifies the previous commitment c_{i-1}. The CAC then sends this message to the reader to be forwarded back to the server. The server can check that the message contents are correct, and then verifies the signatures of the commitments c_{i-1} and c_i against the provided proof pk_{i-1}, raising a detection alert if this validation fails; if it succeeds, both the back-end server and CAC update their state. A detection event can immediately revoke the card's access and trigger the relevant remediation procedure; otherwise the card has successfully authenticated and the appropriate action can follow (e.g. opening the door).

In this scenario, the modified protocol provides both detection of acausal action and contradicting state changes even if the attacker can extract all information from the CAC. In other words, the provided security guarantee is that when an attacker has a cloned copy of a CAC at time t, and used the cloned card at time t', then if the original card has been used in the time interval between t and t', the cloning of the card will be detected. If the original card is not used in the time internal between t and t', then the attacker can use the card to get access, but the cloning attack will be detected as soon as the original card is used again.

Note that while this protocol requires three signing operations on the part of the card, two of these are by temporary commitment keys which only need to remain secure until the next authentication. As such, a weaker and faster signature computation can be used for these to reduce the computation required by the card.

In Section 7.1.4, we will discuss modelling the modified ISO-IEC 9798-3-3 and verify the following properties in our model.

Sound detection. If there is a `detect` event triggered for a particular key pairing, then the responder's key was compromised before the `detect` event.

Guaranteed detection of misuse in prior sessions. Any session violating agreement will be detected during the next session with agreement, even if the adversary is allowed to compromise all state and any one of the keys.

Guaranteed detection of misuse in future sessions. If there was a previous session with agreement and the adversary has not revealed R's state since that session, then any session violating agreement will be detected, even if the adversary has compromised all keys.

4.4 Concluding remarks

In this chapter, we have shown some applications for the foundations we developed in Chapter 3, both in improving existing work and in the design of novel protocols. We showed that a minor modification to transparency overlays could provide stronger security guarantees, and gave two examples of two-party protocols capable of automatically detecting secret misuse without false positives.

We used the first protocol—which detects based on counter state—as a template to modify Keyless SSL, a real-world protocol used by content delivery networks. These modifications introduce little overhead, and provide ideal detection guarantees for the environment where Keyless SSL is applied. For our second protocol, based on 'pre-commitments' to a future session, we conjectured that it achieves even stronger detection guarantees. Such guarantees are ideal for high-security scenarios, and as an example we considered smartcard authentication where it may be desirable to not only detect cloned cards after their use, but also detect them when first used wherever possible.

In order to verify that these protocols achieve the properties stated here—and verify properties of other stateful protocols—we must first improve the tools we have at hand. To that end, we now switch our focus to improving the ability of TAMARIN to analyze stateful protocols. In Chapter 7, we will return to the four novel protocol designs from this chapter and discuss modelling and verifying their security properties.

Chapter 5

The Tamarin Prover

Tamarin is a protocol verification tool that analyzes symbolic protocol models specified in terms of a multiset rewriting system dictating protocol execution combined with a message deduction theory determining how an adversary may manipulate message terms. It can perform both falsification (i.e. finding a counter-example) and unbounded verification of temporal first-order properties with respect to these models. In this chapter we will give a brief overview of the theory implemented by Tamarin that is relevant to our protocol modelling and verification in Chapters 6 and 7. We will also discuss some details specific to the implementation of Tamarin that are necessary to understand how we improve analysis of stateful protocols in Chapter 6. For a more complete discussion of the theory underlying Tamarin we refer the reader to [37, 63, 64, 80].

5.1 Term rewriting

In Tamarin, cryptographic messages and operations are abstracted as terms with rewrite rules. This approach derives many of its methods from the more generic notion of term rewriting, and here we briefly recall some standard notation used (as in, e.g. [42]) and how it applies to Tamarin.

Order-sorted term algebras A signature $\mathbf{\Sigma} = (S, \leq, \Sigma)$ is said to be an *order-sorted signature* if S is a set of sorts, $\leq$ is a partial order on S, Σ is a set of a function symbols associated with sorts, and

1. for every $s \in S$, the connected component of s in $(S, \leq)$ has a top sort $top(s)$,

2. for every $f : s_1 \times \cdots \times s_k \to s$ in Σ with $k \geq 1$, there is a corresponding $f : top(s_1) \times \cdots \times top(s_k) \to s$ in Σ.

In TAMARIN, cryptographic messages are modelled using a set of sorts comprising a top sort *msg* and two mutually incomparable sorts *fr* and *pub* such that both $fr \leq msg$ and $pub \leq msg$. The fresh sort *fr* is used for terms like nonces and random numbers that are freshly generated, while the public sort *pub* is used to model terms like message tags that are known to all participants (including the adversary).

Cryptographic operations are modelled through the choice of an arbitrary set of function symbols Σ as well as a restricted equational theory that we discuss later. For example, TAMARIN's built-in symmetric encryption has a function signature

$$\Sigma = \{senc(\cdot,\cdot), sdec(\cdot,\cdot)\},$$

both of sort $msg \times msg \to msg$.

We assume that there are countably infinite sets of variables $\mathcal{V}_s$ for all sorts $s \in S$, and define $\mathcal{V} = \uplus_{s \in S} \mathcal{V}_s$ the set of all variables. To denote a variable $x \in \mathcal{V}_s$, we write $x : s$. For a signature $\mathbf{\Sigma}$ and a subset of variables $\mathcal{V}' \subseteq \mathcal{V}$, $\mathcal{T}_{\mathbf{\Sigma}}(\mathcal{V}')$ denotes the set of all well-sorted terms constructed over $\Sigma \cup \mathcal{V}'$. For example, using the sorts and symmetric encryption function symbols above, one can construct terms such as $t = senc(x : msg, y : msg)$. We write $\mathcal{T}_{\mathbf{\Sigma}}(\mathcal{V}')_s$ to refer to only those well-sorted terms of sort s.

Subterms A term is constructed of subterms with positions, where a position is a sequence of natural numbers representing the path taken to reach the subterm when viewing the term as a tree. We define a partial ordering relation between two positions p and q such that $p \leq q$ if $\exists p' . (q = p \cdot p')$, i.e. p is a prefix of q (note that p' may be an empty sequence). Positions p and q are called *incomparable* if neither $p \leq q$ nor $q \leq p$. For a term t, we write t_p for the subterm at position p, and $t[u]_{/p}$ for the term t where t_p is replaced by u.

For a term t, the set $Subterms(t)$ is the set of all subterms of t. We define $Var(t) = Subterms(t) \cap \mathcal{V}$, the set of all variables in t. A term t is said to be *ground* if $Var(t) = \emptyset$.

As an example, using the symmetric encryption function symbols above, for terms t, t_1, t_2 where $t = sdec(t_1, t_2)$, the terms t_1 and t_2 are subterms of t at positions $\langle 1 \rangle$ and $\langle 2 \rangle$ respectively. If $t_1 = senc(t', x : msg)$, for a term t' and variable $x : msg$, then

t' is a subterm of t with position $\langle 1, 1\rangle$. Note in this case that the position of t_2 is incomparable with the position of t', and $Var(t) = \{x\!:\!msg\}$.

Equational theory An *equation* over an order-sorted signature $\mathbf{\Sigma}$ is written $s \simeq t$ for terms $s, t \in \mathcal{T}_{\mathbf{\Sigma}}(\mathcal{V})$. A signature $\mathbf{\Sigma}$ and a set of equations E over that signature induce a congruence relation $=_{\mathbf{\Sigma},E}$ over the terms $\mathcal{T}_{\mathbf{\Sigma}}(\mathcal{V})$, which we call an *equational theory*. We identify the equational theory $=_{\mathbf{\Sigma},E}$ with the set of equations E when the signature is clear from context.

Recall the symmetric function symbols above. Equational theories are used to formalize the semantics of cryptographic operations, and the built-in symmetric encryption corresponds to an equational theory

$$E = \{sdec(senc(m, k), k) \simeq m\}.$$

This models the decryption of an encrypted message m using the key k which was used to encrypt it.

TAMARIN includes built-in function symbols with corresponding equational theories for several common cryptographic operations which can be found in [87]. For our examples in this section, we will use the symmetric encryption above as well as those for constructing tuples

$$\Sigma_{pairing} = \{pair(\cdot, \cdot), fst(\cdot), snd(\cdot)\},$$

$$E_{pairing} = \{fst(pair(x, y)) \simeq x, snd(pair(x, y)) \simeq y\}.$$

For simplicity and consistency with our sequence notation, we write $\langle x_1, x_2, \ldots x_n\rangle$ for $pair(x_1, pair(x_2, \ldots pair(x_{n-1}, x_n) \ldots)$.

Substitutions and unification A substitution σ is a map from some set of variables $dom(\sigma) \subseteq \mathcal{V}$ to terms $range(\sigma) \subseteq \mathcal{T}_{\mathbf{\Sigma}}(\mathcal{V})$ such that $x\!:\!s \in dom(\sigma)$ implies $\sigma(x)\!:\!s \in range(\sigma)_s$. By convention, we write $t\sigma$ for a homomorphic extension of σ applied to the term t, and assume that substitutions are idempotent, such that $(t\sigma)\sigma = t\sigma$. A substitution is called a renaming if there exists an inverse (idempotent) substitution σ^{-1} such that $\forall t \in dom(\sigma)\ .\ (t\sigma)\sigma^{-1} = t$. A unifier of two terms s and t over a relation $=_E$ is a substitution σ such that $s\sigma =_E t\sigma$.

For example, a substitution σ which maps a variable $x\!:\!msg$ to the term k is a

unifier of the terms m and $sdec(senc(m, k), x\!:\!msg)$, as

$$sdec(senc(m, k), k) =_E m.$$

Rewriting modulo A rewrite rule over an order-sorted signature $\mathbf{\Sigma}$ is written $l \to r$ for terms $l, r \in \mathcal{T}_{\mathbf{\Sigma}}(\mathcal{V})$. A rewrite system R is a set of rewrite rules defining a relation $\to_R$, such that $s \to_R t$ if there is a rule $l \to r \in R$, a substitution σ, and a position p in s such that $s_p = l\sigma$ and $s[r\sigma]_{/p} = t$. A rewrite system R modulo an equational theory E is defined similarly by a relation $\to_{R,E}$ where instead $s_p =_E l\sigma$ and $s[r\sigma]_{/p} =_E t$.

Tamarin decomposes user-defined equational theories into a rewrite system, which is only possible for a restricted class of equational theories. We will not go into detail about these restrictions or how they help with verification here, as they are not relevant to our work. For detailed discussion see the original description in [81] and recent work relaxing these restrictions in [37].

The rewrite system associated with the equation theory for symmetric encryption above is simply

$$sdec(senc(m, k), k) \to m.$$

Note that only one direction is included, which ensures that the rewrite rules can only be applied a finite number of times to any finite term. This condition is in part what motivates the restrictions on the user-defined equational theories mentioned above (for a formal discussion, see [81]).

5.2 Multiset rewriting in Tamarin

Tamarin uses multiset rewriting rules to specify protocol and adversary execution. Multiset rewriting has been used in the context of protocol verification as early as the formalism introduced by Cervesato et al. [18], which spawned the *MSR* [17] and MuCAPSL [31] languages for specifying security protocols, and is also used in Maude-NPA [41]. For use in security protocols, these approaches extend standard multiset rewriting with support for choosing "new" symbols, as needed for fresh terms. Tamarin extends this further, by additionally supporting facts with arbitrary multiplicity and labels for rewriting rules.

Here we briefly recall some definitions from [63, 80] specifying facts and rules, how they define a trace set, and how trace properties are specified in Tamarin. In the

following, we assume there are two countably infinite sets FN and PN of *fresh* and *public* names respectively, which we will use for assigning names to terms of sort fr and *pub* in evaluating trace properties. We write $\mathcal{T}$ to refer to $\mathcal{T}_{\mathbf{\Sigma}}(FN \cup PN \cup \mathcal{V})$, and $\mathcal{M}$ for $\mathcal{T}_{\mathbf{\Sigma}}(FN \cup PN)$, where $\mathbf{\Sigma}$ will be clear from context.

When discussing multisets we reuse familiar set notation annotated with $\sharp$. For example, we use $\backslash^{\sharp}$ to refer to the multiset extension of the $\backslash$ operation, $\{a, b, b\}^{\sharp}$ for the multiset containing a once and b twice, etc. To refer to the set of elements of a multset M, we write $set(M)$.

5.2.1 Facts

A fact is a *tag* drawn from finite signature $\Sigma_{\mathcal{F}}$ and a sequence of terms from $\mathcal{T}$. Fact tags each contain a name, an arity, and a *multiplicity*. The multiplicity of a fact may be either *linear*, in which case it is removed from state when consumed by a rewrite rule, or *persistent*, remaining in state even if consumed by a rule. Linear facts are used to model, for example, limited resources and mutable state; persistent facts are used to model things like read-only memory, adversary knowledge, and causal relationships. A linear fact Fa/k containing terms $\langle t_1, t_2, \ldots, t_k \rangle$ is written $\mathsf{Fa}(t_1, t_2, \ldots, t_k)$, while persistent facts are denoted in TAMARIN's syntax by prefixing their name with an exclamation mark, e.g. $!\mathsf{Fa}(t_1, t_2, \ldots, t_k)$. We use $\mathcal{F}$ to refer to the set of all facts, and $\mathcal{G}$ to refer to all ground facts (i.e. all terms are ground).

Certain fact tags are given special meaning by Tamarin:

- $\mathsf{Fr}/1$ to model the generation of fresh names,
- $\mathsf{In}/1$ to model receiving from a Dolev-Yao network, and is produced only by adversary construction rules,
- $\mathsf{Out}/1$ to model sending to a Dolev-Yao network, as a premise to adversary deconstruction rules, and
- $!\mathsf{K}/1$ to model adversary knowledge.[1]

In addition to these, we commonly represent agent state with facts prefixed by St to avoid confusion with other facts. We will discuss the interaction of $\mathsf{In}/1$, $\mathsf{Out}/1$, and $!\mathsf{K}$ facts further in Section 5.2.2.

[1]In implementation, $!\mathsf{K}$ facts are split into KU and KD facts in order to prevent redundant message deduction steps during constraint reduction, though this is not relevant to our work. For more information on the distinction between these two facts and how they are used, see [80, Section 3.2.3].

5.2.2 Rewriting rules

A labelled multiset rewriting rule is defined by three fact sequences: its *premises* (or *left-hand side*) l, its *actions* a, and its *conclusions* (or *right-hand side*) r, together written $[\, l \,]\!\!-\!\![\, a \,]\!\!\rightarrow\!\![\, r \,]$. For legibility when l, a, and r are longer sequences of facts, we may write $[\, l_1, \ldots, l_{|l|} \,]\!\!-\!\![\, a_1, \ldots, a_{|a|} \,]\!\!\rightarrow\!\![\, r_1, \ldots, r_{|r|} \,]$ equivalently as

$$\frac{l_1, l_2, \ldots, l_{|l|}}{r_1, r_2, \ldots r_{|r|}}[a_1, a_2, \ldots, a_{|a|}],$$

following the notation for inference rules. In Tamarin, each rule also has some associated *rule information*, including annotations like a rule name that do not affect the semantics of the rule.

The term 'labelled' in *labelled multiset rewriting* refers to labelling of rules with *actions*. The actions of a rule are sequences of facts that are used to express execution properties; we will discuss this further in Section 5.3. All multiset rewriting rules in Tamarin are labelled (though the actions may be an empty sequence), so we generally drop the 'labelled' qualifier and refer to them just as multiset rewriting rules. Finally, given a rule ru we use $prems(ru)$, $acts(ru)$, and $concs(ru)$ to refer to the premises, actions, and conclusions respectively.

The multiset rewriting rules which define a rewrite system in Tamarin are divided into two sets: *protocol rules* and *message deduction rules*. Protocol rules are used to specify both the behaviour of honest participants as well as model-defined adversary capabilities like key reveal. Here, we will give an example of a protocol rule used to one step of the commitment example protocol, before detailing the specific conditions that a rule must meet to be a valid protocol rule in Tamarin.

Example 16 (A protocol rule)**.** To model the first step of the initiator's first step in the commitment example protocol shown in Figure 4.4, we write a rule named '`I_1`' which takes in a fact representing the state of an agent taking role I, St_I, and a fresh nonce value in the premises. The rule rewrites them to an updated state fact and an Out fact to send the first message to the network. Written in Tamarin's syntax, this rule is represented as follows.

```
rule I_1:
    let pkA = pk(~ltkA)
        m1 = <‘1’,pkA, pkB,~ni,prevcommitsig> in
    [ St_I(~id, ~ltkA, pkB,‘m1’,prevcommit,prevcommitsig)
    , Fr(~ni)
    ]
    --[ ]->
    [ St_I(~id, ~ltkA, pkB,<‘m2’,~ni>,prevcommit,prevcommitsig)
    , Out(<m1, sign{m1}~ltkA>)
    ]
```

We omit the actions in the above rule as they are only relevant in the context of lemmas, which we will discuss in the context of the full protocol in Section 7.1.3. The agent's state fact, St_{I}, contains her long term key `ltkA`, the public key of the agent she is communicating with, `pkB`, a term identifying the stage of the protocol she is executing, `‘m1’`, as well as the previous commitment and its signature.

Note that the syntax used for writing TAMARIN rules allows for some shorthand for ease of representation. The symbol ‘~’ is used to denote terms which are of sort fr, i.e. `~ni` is equivalent to $ni : fr$. Further, rules allow for ‘let’ bindings which allow a user to specify a short representation for a term that will be written several times in the rule; in this example, we use `m1` as shorthand for the tuple containing the terms of the first message.

Protocol rules in TAMARIN must meet six conditions (from [80]).

Definition 17. A *protocol rule* is a multiset rewriting rule $[\, l \,]\!\!-\!\![\, a \,]\!\!\rightarrow\!\![\, r \,]$ such that

P1. l, a, and r do not contain fresh names,

P2. l does not contain $\mathsf{!K}$ and Out facts,

P3. r does not contain $\mathsf{!K}$, In and Fr facts,

P4. the argument of a Fr-fact is always of sort fr

P5. r does not contain the function symbol $*$, and

P6. $[\, l \,]\!\!-\!\![\, a \,]\!\!\rightarrow\!\![\, r \,]$ satisfies both (a) $vars(r) \subseteq vars(l) \cup \mathcal{V}_{pub}$ and (b) l only contains irreducible function symbols from $\Sigma \setminus \Sigma_{\mathcal{DH}}$ (or it is an instance of a rule that does).

Condition **P1** ensures that fresh names are not created directly by protocol rules; instead, fresh names are accessed through the Fr fact generated by a special rule. **P2**

and **P3** ensure that special facts occur only in their intended places. **P4** is not strictly necessary, as Fr facts can only be generated in a particular way and always contain a fresh name, but simplifies later definitions. Condition **P5** restricts Diffie-Hellman product terms from the Diffie-Hellman function signature (denoted $\Sigma_{\mathcal{DH}}$ in **P6**) and is not relevant to our work, but included for completeness. Finally, **P6** prevents rules from referencing non-public names which do not exist in their premises; note that for our purposes we will ignore the Diffie-Hellman functions and thus $\Sigma \setminus \Sigma_{\mathcal{DH}}$ can be considered equivalent to Σ.

To see why **P6** requires that l contain only irreducible function symbols, consider the rule $[\,\mathsf{In}(sdec(senc(m,k),k))\,] \!\!-\!\![\ ]\!\!\rightarrow [\,\mathsf{Out}(k)\,]$ under an equational theory which includes $sdec(senc(m,k),k) = m$. This rule is equivalent to the rule $[\,\mathsf{In}(m)\,] \!\!-\!\![\ ]\!\!\rightarrow [\,\mathsf{Out}(k)\,]$, and thus could send arbitrary terms.

Message deduction rules (also called *intruder rules*) determine how terms may be manipulated by the network adversary to create In facts.

Definition 18. The basic message deduction rules are defined by the union of

$$
\begin{aligned}
pub\colon &\quad [\ ] \!-\![\ ]\!\rightarrow [\,!\mathsf{K}(x : pub)\,] \\
gen_fresh\colon &\quad [\,\mathsf{Fr}(x : fr)\,] \!-\![\ ]\!\rightarrow [\,!\mathsf{K}(x : fr)\,] \\
cf\colon &\quad [\,!\mathsf{K}(x_1) \ldots !\mathsf{K}(x_k)\,] \!-\![\ ]\!\rightarrow [\,!\mathsf{K}(f(x_1,\ldots,x_k))\,] \text{ for all } f \in \Sigma \\
isend\colon &\quad [\,!\mathsf{K}(x)\,] \!-\![\,\mathsf{K}(x)\,]\!\rightarrow [\,\mathsf{In}(x)\,] \\
irecv\colon &\quad [\,\mathsf{Out}(x)\,] \!-\![\ ]\!\rightarrow [\,!\mathsf{K}(x)\,]
\end{aligned}
$$

The adversary knowledge is represented by $!\mathsf{K}()$ facts. The rules *pub* and *gen_fresh* model the ability of the adversary to learn any public name and generate their own fresh names respectively. The *cf* rule allows an adversary to combine terms by applying functions. Finally, *isend* and *irecv* allow the adversary to receive and send messages respectively. Note that *isend* is also labelled with $!\mathsf{K}(x)$ so that messages sent by the adversary appear in the protocol trace.

In addition to the basic message deduction rules, TAMARIN currently implements several extensions to these rules, including Diffie-Hellman rules as described in [64] and rules generated to support a larger set of equational theories as described in [37].

For a set of rules R, we write $insts(R)$ and $ginsts(R)$ for the set of instances and ground instances of R respectively. In addition to the rules that make up a rewrite system, we also define a special rule which is the unique source of Fr facts,

$$\text{FRESH} = [\ ]\!\!-\!\![\ ]\!\!\rightarrow\!\![\ \mathsf{Fr}(x\!:\!fr)\ ].$$

The FRESH rule is neither a protocol rule nor a message deduction rule, and is not part of the rewrite system itself. Instead, we include in specially within the semantics of protocol execution, as we see below.

5.2.3 Execution

The state of the transition system is modelled as a finite multiset of facts, which is modified step by step by executing rules.

A valid step of a rewrite system R with respect to an equational theory E is defined by the transition relation $\Rightarrow_{R,E}$:

$$\frac{l\!-\![\,a\,]\!\!\rightarrow r \in_E ginsts(R \cup \{\text{FRESH}\}) \quad lfacts(l) \subseteq^\sharp S \quad pfacts(l) \subseteq set(S)}{S \overset{set(a)}{\Longrightarrow}_{R,E} ((S \setminus^\sharp lfacts(l)) \cup^\sharp mset(r)}$$

where $lfacts(l)$ and $pfacts(l)$ respectively denote the linear and persistent facts in l. We use $\sharp$ to indicate the multiset extension of a set operation, i.e. $\setminus^\sharp$ and $\subseteq^\sharp$ denote the multiset extensions of $\setminus$ and $\subseteq$ respectively. We use $r \in_E R$ to denote that r is an element of R modulo the equational theory E. This transition can apply ground instances of rules if their premises are included in the state S, consumes the linear facts in the premises, and adds the rule conclusions to the state.

From this we can define the set of all traces generated by the multiset rewriting system R modulo an equational theory E in terms of the labels appearing in rules.

$$\begin{aligned} traces_E(R) := \Big\{ \langle A_1, \dots A_n \rangle \ \mid\ & \exists S_1, \dots, S_n \,.\, \emptyset \overset{A_1}{\Longrightarrow}_{R,E} S_1 \overset{A_2}{\Longrightarrow}_{R,E} \dots \overset{A_n}{\Longrightarrow}_{R,E} S_n \\ & \wedge \forall i, j, x \,.\, (i \neq j \wedge (S_{i+1} \setminus^\sharp S_i) = \{Fr(x)\}^\sharp) \Rightarrow \\ & \qquad (S_{j+1} \setminus^\sharp S_j) \neq \{Fr(x)\}^\sharp \Big\}. \end{aligned}$$

The final condition enforces that each instance of FRESH is used at most once in a trace, thereby ensuring that each fresh name is unique.

Example 19 (Simple protocol example)**.** Consider a simple protocol,

$$P_{ex} = \left\{ \frac{\mathsf{Fr}(n : fr), \mathsf{Fr}(k : fr)}{\mathsf{St}(n : fr, k : fr), \mathsf{Out}(\langle n : fr, k : fr \rangle)} [\mathsf{Start}()], \frac{\mathsf{St}(n, k), \mathsf{In}(senc(n, k))}{} [\mathsf{End}()] \right\}.$$

An execution of this protocol is shown in Figure 5.1, which has the trace

$$\langle \{\emptyset, \emptyset, \{\mathsf{Start}()\}, \emptyset, \emptyset, \emptyset, \emptyset, \{\mathsf{K}(senc(n, k))\}, \{\mathsf{End}()\} \rangle.$$

We show the rule instances on the left, with causal dependencies shown as arrows, and the global state on the right. The causal dependencies allow us to determine which sequences of rule instances correspond to valid executions, as each premise must be preceded by a conclusion fact

Note that the sequence of rule instances is sufficient to characterize the execution, as both the trace and the sequence of global states can be reconstructed from it.

5.3 Trace properties

Tamarin makes use of a first-order logic with sorts to specify security properties. Temporal properties are supported through a sort *temp* for timepoints, with variable drawn from $\mathcal{V}_{temp}$. Properties are specified as first-order formulas over *trace atoms*, originally comprising

1. term equality, e.g. $t_1 \approx t_2$,
2. timepoint ordering, e.g. $i \lessdot j$,
3. timepoint equality, e.g. $i \doteq j$,
4. and actions f @ i.

These are extended with a last trace atom to perform trace induction, which we will discuss in Section 6.2.1. In order to evaluate trace formulas, we associate a *valuation* to variables. To ensure valuations respect sorts, we associate a valid domain $\mathbf{D}_s$ to each sort s: $\mathbf{D}_{temp} \subseteq \mathbb{Q}$, $\mathbf{D}_{fr} \subseteq \mathsf{FN}$, $\mathbf{D}_{pub} \subseteq \mathsf{PN}$, and $\mathbf{D}_{msg} \subseteq \mathcal{M}$. A valuation itself is a function θ from $\mathcal{V}$ to $\mathbb{Q} \cup \mathcal{M}$ such that $\theta(V_s) \subseteq \mathbf{D}_s$ for every sort, i.e. it respects these domains. As with substitutions, we write $t\theta$ for the homomorphic extension of θ applied to a term t. Recall that for a sequence s we write $idx(s)$ for the set of all

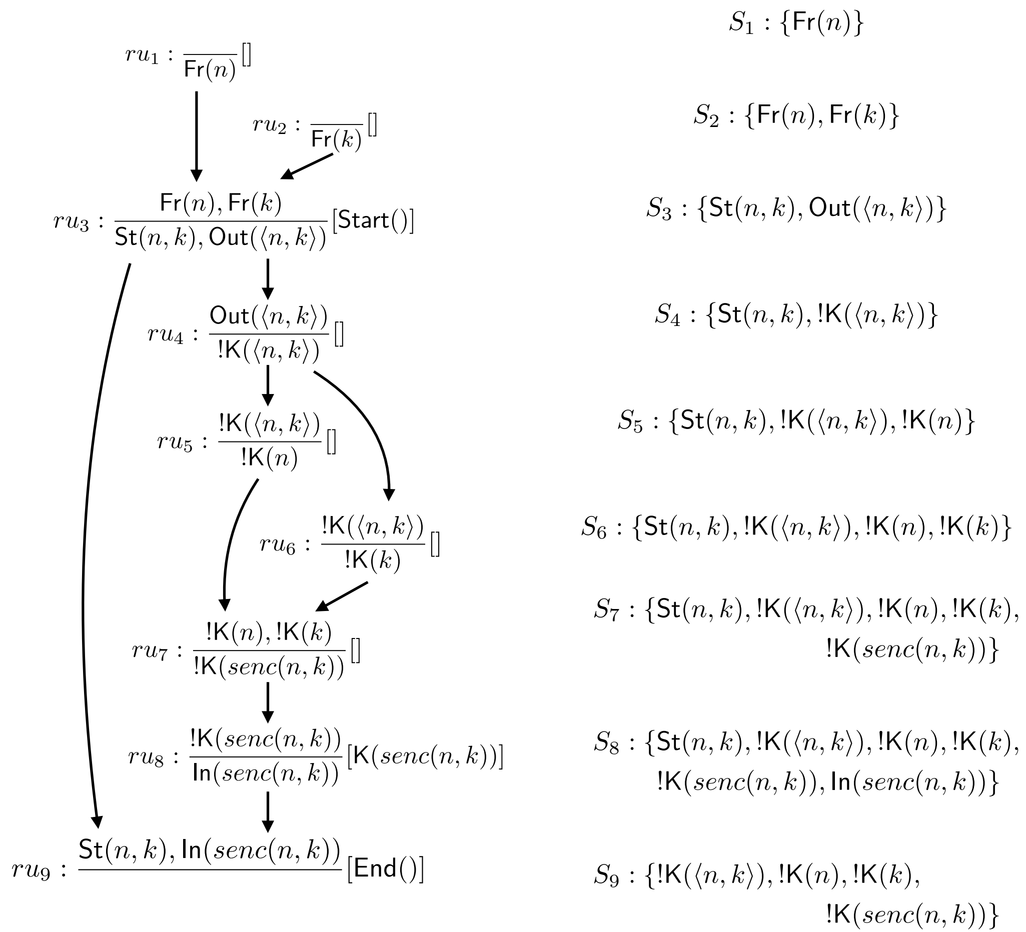

Figure 5.1: An execution of the protocol P_{ex}.

indices of s.

For an equational theory E, a satisfaction relation is defined, relating a trace tr and a valuation θ with a trace formula φ:

$$
\begin{array}{ll}
(tr,\theta) \vDash_E \bot & \text{never} \\
(tr,\theta) \vDash_E f \ @\ i & \text{iff } \theta(i) \in idx(tr) \wedge f\theta \in_E tr_{\theta(i)} \\
(tr,\theta) \vDash_E i \lessdot j & \text{iff } \theta(i) < \theta(j) \\
(tr,\theta) \vDash_E i \doteq j & \text{iff } \theta(i) = \theta(j) \\
(tr,\theta) \vDash_E t_1 \approx t_2 & \text{iff } t_1\theta =_E t_2\theta \\
(tr,\theta) \vDash_E \neg\varphi & \text{iff } \neg\left((tr,\theta) \vDash_E \varphi\right) \\
(tr,\theta) \vDash_E \varphi \wedge \psi & \text{iff } \left((tr,\theta) \vDash_E \varphi\right) \wedge \left((tr,\theta) \vDash_E \psi\right) \\
(tr,\theta) \vDash_E \exists x{:}s\ .\ \varphi & \text{iff } \exists u \in \mathbf{D}_s\ .\ (tr, \theta[x \mapsto u]) \vDash_E \varphi.
\end{array}
$$

The semantics of the logical connectives $\vee$, $\Rightarrow$ and quantifier $\forall$ are defined in the usual way by translating them to the fragments above (e.g. $\varphi \vee \psi$ is translated to $\neg(\neg\varphi \wedge \neg\psi)$).

We extend this relation over trace sets and say that a trace formula φ is *valid* for a trace set Tr modulo E if and only if $(tr,\theta) \vDash_E \varphi$ for every trace $tr \in Tr$ and every valuation θ, and that a trace formula φ is *satisfiable* for a trace set Tr modulo E if and only if there exists a trace $tr \in Tr$ and valuation θ such that $(tr,\theta) \vDash_E \varphi$. We write $Tr \vDash_E^\forall \varphi$ and $Tr \vDash_E^\exists \varphi$ for validity and satisfiability respectively. Importantly, $Tr \vDash_E^\forall \varphi$ if and only if $\neg(Tr \vDash_E^\exists \neg\varphi)$, which allows us to convert from solving validity of a trace property to solving satisfiability of the negated property.

Example 20 (A trace property)**.** To express a property of P_{ex} such as 'an $\mathsf{End}()$ action is preceded by a $\mathsf{Start}()$ action,' we can write a trace formula φ_{ex} where

$$\varphi_{ex} = \exists i{:}temp\ .\ \mathsf{End}()\ @\ i \Rightarrow \exists j{:}temp\ .\ (j \lessdot i) \wedge \mathsf{Start}()\ @\ j.$$

If this property is true in all traces, then $traces_E(P_{ex}) \vDash_E^\forall \varphi_{ex}$; if it is not, then $traces_E(P_{ex}) \vDash_E^\exists \neg\varphi_{ex}$.

5.4 Constraint solving

Satisfiability claims are proved or disproved through a constraint solving algorithm, which incrementally builds a satisfying trace through a series of constraint reduction steps. Validity claims can also be solved using the same process, by proving or disproving satisfiability of the negated claim. The constraint solving algorithm exhaustively searches for satisfying traces by beginning with the constraints introduced by a claim and solving these through a series of constraint reduction steps. In this section we will first discuss the dependency graphs used to represent solutions to constraint systems before showing how claims are solved. For brevity we omit the many motivating examples given in prior work; for a reader wishing to build further intuition we recommend [63].

Throughout this section, we use a convention for variable names. We use f to range over facts, i and j over temporal variables, ri over multiset rewriting rule instances, and u and v over natural numbers, unless otherwise specified. This is consistent with the convention used in [63, 80].

5.4.1 Dependency graphs

Tamarin makes use of dependency graphs to represent protocol executions in a way that captures causal dependencies between the rules. Dependency graphs comprise nodes, labelled with rule instances, and edges representing the dependencies between nodes. An example of a dependency graph is shown in Figure 5.2, corresponding to the execution of the protocol P_{ex} shown in Figure 5.1. The edges show causal dependencies between nodes: an edge from a conclusion fact of one node to a premise fact of another marks that a fact is generated by the former and consumed by the latter. Persistent facts, prefixed with !, are allowed to have multiple outgoing edges. Note that the rules at each node are ground instances in our equational theory; for example, $!\mathsf{K}(n)$ in the first conclusion of ru_5 is equivalent to $!\mathsf{K}(fst(\langle n, k\rangle))$ modulo $E_{pairing}$.

Formally, Meier defines dependency graphs in [63, Section 8.1] as follows.

Definition 21. For an equational theory E and a multiset rewriting system R, $dg := (I, D)$ is an R, E-dependency graph if the nodes I are a sequence of ground instances of $R \cup \{\text{Fresh}\}$ (i.e. $\forall i \in idx(I) \;.\; I_i \in ginsts_E(R \cup \{\text{Fresh}\})$), the edges $D \subseteq \mathbb{N}^2 \times \mathbb{N}^2$, and dg satisfies the conditions **DG1-4** below. In the following, we refer

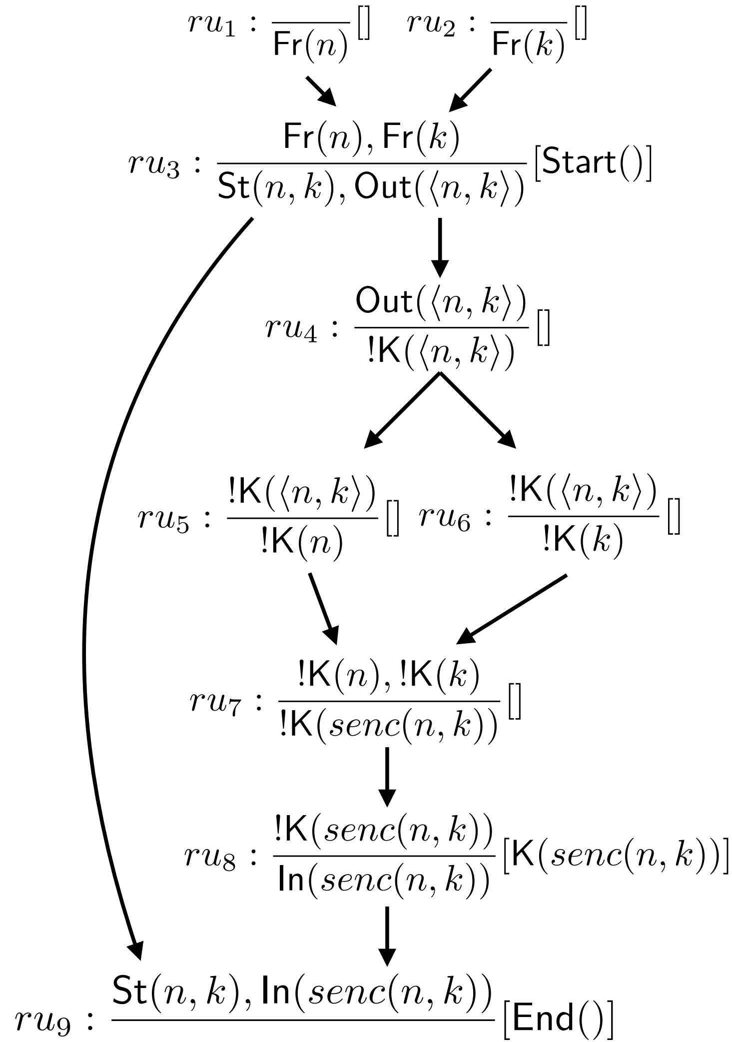

Figure 5.2: The dependency graph corresponding to the execution of P_{ex} in Figure 5.1, where ru_1 through ru_9 are indices. Each edge depicts a causal relation between nodes, and is identified by the indices of the conclusion and premise linked by the edge. For example, the edge between the first conclusion of ru_3 and the first premise of ru_9 is $((3,1),(9,1))$.

to the *premises* and *conclusions* of dg as pairs (i, u) such that i is a node of dg and $u \in idx(prems(I_i))$ or $u \in idx(concs(I_i))$ respectively. These have associated *premise* and *conclusion facts* $prems(I_i)_u$ and $concs(I_i)_u$ respectively. Thus, each fact instance can be represented by a pair of integers: the index of its corresponding rule instance, and the index of the premise or conclusion within that rule.

DG1 for every edge $((i, u), (j, v)) \in D$, $i < j$, and the conclusion fact of (i, u) is equal modulo E to the premise fact of (j, v);

DG2 every premise of dg has exactly one incoming edge;

DG3 every conclusion of dg with a linear conclusion fact has at most one outgoing edge;

DG4 instances of Fresh in I are unique.

The trace of a dependency graph (I, D) where $I := \langle ri_0, \ldots, ri_n \rangle$ is defined as $\langle set(acts(ri_0)), \ldots, set(acts(ri_n)) \rangle$. The set of all R, E-dependency graphs is denoted $dgraphs_E(R)$.

In order to use dependency graphs as an alternative formulation of the multiset rewriting semantics above, it is necessary to prove equivalence between the two. Meier proves the following theorem.

Theorem 22. *For every multiset rewriting system R and every equational theory E,*

$$traces_E(R) =_E \{trace(dg) \mid dg \in dgraphs_E(R)\}.$$

Dependency graphs in TAMARIN are also extended to support message deduction over associative and commutative equational theories like Diffie-Hellman (see [80, Section 3.2.3]) though this is not relevant to our work.

5.4.2 Guarded trace properties

In order to ensure we can solve a trace property, we must restrict the set of allowed properties. We must ensure that they are closed under negation so that we can convert between satisfiability and validity. Further, we must ensure that when evaluating the trace formula it is sufficient to consider only the subterms that appear in a given trace. To do this, we first define the notion of *guarded trace property*, a class of trace properties which meet these restrictions.

Definition 23. A trace formula φ is a *guarded trace formula* if all logical operators are $\wedge, \vee, \forall$, and $\exists$, negation is applied only to trace atoms and $\perp$, and all of its quantifiers are of the form $\exists x \,.\, g \wedge \psi$ or $\forall x \,.\, \neg g \vee \psi$ such that both

G1. $x \subseteq \mathcal{V}_{msg} \cup \mathcal{V}_{temp}$, and

G2. either

(a) g is an action $f \ @ \ i$ such that $x \subseteq vars(f \ @ \ i)$, or

(b) g is an equality $s \approx t, vars(s) \cap x = \emptyset$ such that $x \subseteq vars(t)$.

A guarded trace formula φ is a *guarded trace property* if it is closed and for all terms t occurring in φ, t is a variable or a public name.

In general, Tamarin can automatically convert most trace properties that a modeller may write to guarded trace properties, so it is not typically necessary to keep these requirements in mind when writing a model. They are, however, necessary for the constraint solving performed by Tamarin.

Constraints are used to indicate necessary properties of a dependency graph. A constraint is either

1. a guarded trace formula,
2. a node constraint, written $i : ri$ for an index i and rule instance ri,
3. a premise constraint, written $f \blacktriangleright_v i$ for a fact f occurring as the vth premise at index i, or
4. an edge constraint, written $(i, u) \rightarrowtail (j, v)$ for an edge between the uth conclusion at index i and the vth premise at index j.

These are used to represent restrictions that must be met by a dependency graph to be a satisfying trace. A guarded trace formula indicates that the guarded trace formula must be satisfied by the trace of the graph. A node constraint $i : ri$ indicates that a rule instance ri must occur at node i. A premise constraint $f \blacktriangleright_v i$ indicates that a fact f must occur as the vth premise of the rule instance at node i. An edge constraint $(i, u) \rightarrowtail (j, v)$ indicates that there must be an edge between the uth conclusion at index i and the vth premise at index j.

A *constraint system* is a finite set of constraints. We define a constraint satisfaction relation between a dependency graph $dg = (I, D)$ with valuation θ and a constraint γ,

written $(dg,\theta) \Vdash_E \gamma$, as

$$
\begin{aligned}
&(dg,\theta) \Vdash_E i : ri && \text{iff } \theta(i) \in idx(I) \text{ and } ri\theta =_E I_{\theta(i)} \\
&(dg,\theta) \Vdash_E f \blacktriangleright_v i && \text{iff } \theta(i) \in idx(I) \text{ and } f\theta =_E prems(I_{\theta(i)})_v \\
&(dg,\theta) \Vdash_E (i,u) \rightarrowtail (j,v) && \text{iff } ((\theta(i),u),(\theta(j),v)) \in D \\
&(dg,\theta) \Vdash_E \varphi && \text{iff } trace(dg,\theta) \vDash_E \varphi.
\end{aligned}
$$

For a constraint system Γ, we extend this relation such that $(dg,\theta) \Vdash_E \Gamma$ iff

$$\forall \gamma \in \Gamma \,.\, (dg,\theta) \Vdash_E \gamma.$$

A R,E-solution of a constraint system Γ is an R,E-dependency graph dg such that there is a valuation θ with $(dg,\theta) \Vdash_E \Gamma$. Thus, a dependency graph satisfies a constraint system if there is some valuation which allows it to satisfy all constraints in the constraint system.

Example 24 (Constraint satisfaction)**.** An edge constraint $(15,1) \rightarrowtail (20,2)$ is satisfied by the dependency graph shown in Figure 5.2, e.g. through

1. a valuation θ_1 where $\theta(15) = 6$ and $\theta(20) = 7$,
2. a valuation θ_2 where $\theta(15) = 2$ and $\theta(20) = 3$, or
3. a valuation θ_3 where $\theta(15) = 8$ and $\theta(20) = 9$.

A constraint system $\{(15,1) \rightarrowtail (20,2), \mathsf{St}(n,k) \blacktriangleright_1 20\}$ is also satisfied by the same dependency graph, using the valuation θ_3 above.

A constraint system $\{(15,1) \rightarrowtail (20,2), \mathsf{St}(n,k) \blacktriangleright_1 15\}$, however, is not satisfied by this dependency graph, as the only node for which $\mathsf{St}(n,k)$ occurs as the first premise is the node at index 9, but this node has no outgoing edge (or any conclusion at all). Both constraints cannot be satisfied simultaneously by any valuation.

5.4.3 Constraint reduction

The intuition behind the constraint solving algorithm used by TAMARIN is to build a constraint-reduction relation $\rightsquigarrow_{R,E}$ which refines the initial constraint system defined by a guarded trace property into a set of refined constraint systems. This is done until either a dependency graph with a valuation satisfying a refined constraint system is found, or all resulting constraint systems contain contradictory constraints.

The constraint reduction relation comprises three parts. *Case distinction* rules add additional constraints by splitting a constraint system into a set of constraint systems representing possible cases. *Simplification* rules are rules which do not introduce case distinctions. *Contradiction* rules can remove constraint systems entirely, when there are contradictory constraints in the system. Each constraint reduction rule in the relation must be proven to be both *complete* and *sound*, i.e. for a rule $\Gamma \rightsquigarrow_{R,E} \{\Gamma_1, \ldots, \Gamma_n\}$, a R, E-solution of Γ is a R, E-solution of at least one of the constraint systems $\{\Gamma_1, \ldots, \Gamma_n\}$, and vice versa respectively.

Figure 5.3 shows a few examples of constraint reduction rules in the basic constraint reduction relation $\rightsquigarrow_{R,E}$ (see Meier [63, Figure 8.3] for a complete list). The $\texttt{DG}_{\lessdot}$ is an example of a contradiction rule, which reduces a constraint system containing $i \lessdot i$ (i.e. that a timepoint is less than itself) to $\perp$. The $\texttt{DG}_{\blacktriangleright}$ rule is an example of a simplification rule: it does not introduce any additional constraint systems, simply adds to the existing constraint system the corresponding premise constraints for each premise in a rule. Finally, the $\texttt{S}_{\blacktriangleright}$ rule is an example of a case distinction rule: it 'solves' a premise constraint by splitting a constraint system into several systems, one for each possible conclusion that might be the source of a premise constraint. We will give an example of applying the latter two rules to a specific constraint system in the context of precomputation in the following section.

The boxed portion of the rules indicates the conditions that exclude the rule from being applied redundantly; for example, $\texttt{DG}_{\blacktriangleright}$ will not introduce a premise constraint when one already exists in the constraint system. In [63, Section 8.2.3], Meier proves that all rules forming the basic constraint reduction relation are sound and complete, and that from every solved constraint system it is possible to extract a solution trace.

Note that TAMARIN implements more constraint reduction rules than found in [63, Figure 8.3]. In Section 6.2.1, we describe an extension to the basic constraint reduction relation and a trace property transformation that allow for a form of trace induction, and it is also necessary to extend them to support the message deduction in equational theories like Diffie-Hellman. The latter is not relevant to our work, but can be found in [63, Section 8.4].

$$\texttt{DG}_{\lessdot}: \quad \Gamma \rightsquigarrow \bot \qquad \text{if } i \lessdot \; i$$

$$\texttt{DG}_{\blacktriangleright}: \quad \Gamma \rightsquigarrow (f \blacktriangleright_v i, \Gamma) \qquad \text{if } ((i,v), f) \in ps(\Gamma); \; \boxed{(f \blacktriangleright_v i) \notin_E \Gamma}$$

$$\texttt{S}_{\blacktriangleright}: \quad \Gamma \rightsquigarrow \|_{ri \in R \cup \{\textsc{Fresh}\}} \|_{u \in idx(concs(ri))} (i : ri, \; (i,u) \rightarrowtail (j,v), \; \Gamma)$$

$$\text{if } \underline{(f \blacktriangleright_v j)} \in \Gamma \text{ and } i \text{ fresh}; \; \boxed{\text{there is no } c \text{ such that } (c \rightarrowtail (j,v)) \in \Gamma}$$

Figure 5.3: An example of three of the constraint reduction rules which form the $\rightsquigarrow_{R,E}$ relation from [63, Figure 8.3]. $\Gamma \rightsquigarrow \Gamma_1 \| \ldots \| \Gamma_n$ is written for an n-fold case distinction, i.e. $\Gamma \rightsquigarrow \{\Gamma_1, \ldots, \Gamma_n\}$, and $\Gamma \rightsquigarrow \Gamma'$ abbreviates $\Gamma \rightsquigarrow \{\Gamma'\}$. We use $ps(\Gamma)$ as shorthand for all premises of node constraints in Γ, i.e. $\{((i,v), prems(ri)_v) \mid i : ri \in \Gamma, v \in idx(prems(ri))\}$.

5.5 Implementation and precomputed sources

Proofs in Tamarin are performed by incrementally solving applicable goals until either all goals are solved (and thus a satisfying trace is found) or the initial constraint system has been reduced to $\bot$. Both contradiction and simplification rules are applied eagerly by Tamarin at each step between solving goals, and the proof is thus represented by the series of goals solved and contradictions applied, omitting simplification rules that would make proofs difficult to read.

Goals are chosen at each step according to a heuristic function, which attempts to assign a useful priority order to different goals. We will discuss heuristics further, including the priorities chosen by the default heuristic, and related issues in Section 6.1.

Tamarin also performs several steps of precomputation on a protocol model prior to analyzing trace properties. One of these involves computing rule variants necessary to support AC equational theories like Diffie-Hellman, which is not relevant to our work; we refer an interested reader to [63, Section 3.2.2] for more information about what is involved and why it is necessary. In addition to this, Tamarin computes a set of rule premises called *loop breakers*, used in later precomputation as well as in heuristic distinctions that we discuss in Section 6.1.2.

Finally, Tamarin also precomputes a limited set of case distinction rules. This precomputation simplifies some common constraint-reduction steps that are applied repeatedly in a protocol, contracting several constraint reductions into one proof step. Specifically, this step precomputes the resulting constraint systems from solving a *premise goal*, for all facts that appear as premises in protocol rules and message deduction rules (see `Sources.hs`, [85]).

Formally, for every fact $f(\vec{x})$ that occurs as a rule premise, TAMARIN begins with the constraint system $\{f(\vec{x}) \blacktriangleright_1 i : temp\}$, applies the relevant constraint reduction rule $\mathtt{S}_\blacktriangleright$, and applies a limited set of further constraint reduction rules (e.g. not solving loop breakers or goals related to message deduction, see [80, Section 3.3.4]). These constraints are solved until TAMARIN is left with a set of constraint systems representing all possible 'sources' of the premise—a refined set of all of the constraint systems that could be created from solving the premise goal. Precomputing sources of premises in this way also reveals which premise goals *do not* lead to multiple cases, allowing them to be applied eagerly in the future.

Example 25 (Precomputed source)**.** To precompute the sources of $\mathsf{St}(n, k)$ in the protocol P_{ex}, TAMARIN begins with the constraint system $\{\mathsf{St}(n, k) \blacktriangleright_1 i : temp\}$. In this case, the premise constraint reduction $\mathtt{S}_\blacktriangleright$ is applied to solve the premise constraint, leading to a new constraint system

$$\{\mathsf{St}(n, k) \blacktriangleright_1 i, \frac{\mathsf{Fr}(n), \mathsf{Fr}(k)}{\mathsf{St}(n, k), \mathsf{Out}(\langle n, k\rangle)}[\mathsf{Start}()] : j_1, (j_1, 1) \rightarrowtail (i, 1)\},$$

where $i, j_1 : temp$. Applying $\mathtt{DG}_\blacktriangleright$ twice adds the two $\mathsf{Fr}()$ premise constraints, leading to the constraint system

$$\Big\{\mathsf{St}(n, k) \blacktriangleright_1 i, \frac{\mathsf{Fr}(n), \mathsf{Fr}(k)}{\mathsf{St}(n, k), \mathsf{Out}(\langle n, k\rangle)}[\mathsf{Start}()] : j_1, (j_1, 1) \rightarrowtail (i, 1), \mathsf{Fr}(n) \blacktriangleright_1 j_1, \\ \mathsf{Fr}(k) \blacktriangleright_2 j_1\Big\}.$$

Finally, solving each of these $\mathsf{Fr}()$ premise constraints with the $\mathtt{S}_\blacktriangleright$ reduction results in the constraint system

$$\Big\{\mathsf{St}(n, k) \blacktriangleright_1 i, \frac{\mathsf{Fr}(n), \mathsf{Fr}(k)}{\mathsf{St}(n, k), \mathsf{Out}(\langle n, k\rangle)}[\mathsf{Start}()] : j_1, (j_1, 1) \rightarrowtail (i, 1), \mathsf{Fr}(n) \blacktriangleright_1 j_1, \\ \mathsf{Fr}(k) \blacktriangleright_2 j_1, \frac{}{\mathsf{Fr}(n)}[] : j_2, (j_2, 1) \rightarrowtail (j_1, 1), \frac{}{\mathsf{Fr}(k)}[] : j_3, (j_3, 1) \rightarrowtail (j_1, 2)\Big\}.$$

This constraint system contains the source of $\mathsf{St}(n, k)$, the rule containing $\mathsf{Start}()$, as well as the sources of the two fresh terms used in that rule.

To use these precomputed sources, TAMARIN checks whether a goal it is solving has precomputed sources, and if it does, unifies the source systems directly rather than applying the single constraint reduction step. Thus, many constraint reduction

steps may be contracted into a single case distinction.

Though precomputation of sources is done only for premise goals currently, they are implemented generically so that any goal might have sources associated to it. In Section 6.2.2, we discuss work modifying the set of precomputed sources, and our own work precomputing sources for an expanded set of goals to allow Tamarin to account for complex looping behaviour in a protocol during source precomputation.

Chapter 6

Improving automated analysis of stateful protocols

In order to improve Tamarin's ability to analyze stateful protocols, we take several approaches. We begin by improving the heuristics used by Tamarin, with two specific contributions. First, we extend the syntax of facts and rules in Tamarin, allowing a model to include *annotations* that do not change the semantic properties of the model; we use these to implement heuristic annotations which allow for localized modification of goal priority in all existing heuristics. Second, we implement a new heuristic tailored specifically for analysis of stateful protocols.

Next, we examine the trace induction used by Tamarin and discuss its limitations in the stateful setting. From this, we develop modelling techniques which make analysis tractable for complicated protocols with complex invariants, by explicitly capturing invariants as an addition to the model semantics. Though these modelling techniques are powerful, any change to the semantics of a model is not ideal, and the complexity introduced when making these changes to the model may lead to errors. To remove this requirement, we modify the precomputation steps performed by Tamarin to allow so-called `sources` lemmas (formerly *typing lemmas*, see [63, p. 8.4.4]) to be applied more effectively. This modification to Tamarin allows us to write sources lemmas that provide equivalent benefits to protocol analysis, while ensuring that the proof burden for the relevant properties can be discharged by Tamarin with no modification to the model's semantics. An additional complication of trace induction is the necessity of introducing irrelevant action goals, and we discuss a technique for minimizing their impact.

6.1 Improving heuristics

When there are many open goals in a constraint system, the choice of which to solve next is made by sorting the goals according to a heuristic ranking. This can have a large impact on the resulting proof size (and thus performance), as some goals may lead to an irrelevant disjunction of many constraint systems that will all require the same subproof. TAMARIN implements several heuristics for picking the optimal goal to solve, in addition to allowing an external program to sort goals (the so-called 'oracle' heuristic). One of these heuristics is an ordering by age of the goal—without considering any properties of the goals themselves—but the rest are all modified versions of the default heuristic, the '`s`' (or 'smart') heuristic.

In implementing a tailored heuristic for stateful protocol analysis, we also base our heuristic on the '`s`' heuristic, but go further to address a substantial limitation of the '`s`' heuristic caused by its structure. In this section we will first discuss precisely what the '`s`' heuristic does and its limitations, before describing the custom heuristic we implemented and the design choices involved.

6.1.1 Limitations of the smart heuristic

The '`s`' heuristic defines a series of sorts done to the list of goals, with each sort comprising several categories of goals and an ordering of those categories. In each of these sorts, the goals are sorted by the first category they are a part of, so that all goals matching the first category occur before all goals that do not, and so on. The sorts are, in order,

1. an initial sort of goals by their goal number, representing the age of the goal;
2. a sort into priority buckets based on twelve categories of goals, where these categories are (in order of their priority defined by the `s` heuristic):
    (a) deconstruction chain goals,
    (b) disjunction goals,
    (c) premise and action goals with names beginning in '`F_`',
    (d) premise goals,
    (e) action goals that are not adversary deduction,
    (f) action goals for adversary deduction of a private function,

(g) action goals for adversary deduction of a variable x such that $x \in \mathcal{V}_{fresh}$

(h) deferred case splits that result in less than three cases,

(i) action goals for adversary deduction which have exactly one precomputed source,

(j) action goals for adversary deduction of a term x such that $\exists m, k \,.\, x = \mathtt{sign}(m, k)$,

(k) action goals for adversary deduction of a term x such that $\exists g, y_1, y_2 \,.\, x = g^{y_1 * y_2}$,

(l) and finally, all other goals that are *not* deferred case splits;

3. a sort which delays all premise and action goals with names beginning in 'L_' such that they are at the end of the list;

4. and finally, a sort of the goals according to their 'usefulness', which delays

   (a) premise goals for facts marked as loop breakers, and action goals for adversary deduction of a term that does not contain any fresh names or private functions (these goals are referred to as 'probably constructible'),

   (b) and further delays action goals for adversary deduction of a term which contains only public names or a term which can be derived from an Out() fact using only unpairing or inversion (these goals are referred to as 'currently deducible'.)

The broad goal of the heuristic is to solve goals that are either trivial or likely to result in a contradiction. Collectively, the sorts other than the third have a broadly similar result to the original heuristic described by Meier in [63, Section 9.2.3], though more categories have been added and their relative priorities altered. The third sort was motivated by the desire to allow a modeller to 'de-prioritize' certain premise and action goals.

Note that the heuristic provides a means to prioritize (and de-prioritize) particular facts, based on the name of the fact. Though this is helpful, using the name of the fact to determine solving priority has substantial limitations: for two different fact instances to unify, they must have the same name. Thus to prioritize, e.g., a particular premise goal, a modeller necessarily prioritizes *all* premise goals related to that fact. In Section 6.1.2, we introduce an extension to TAMARIN's syntax that allows for *localized*

fact annotations, and modify TAMARIN's heuristics to allow facts to be prioritized or de-prioritized with specific fact annotations.

A more significant limitation found in all the heuristics based on the '`s`' heuristic—that is, all heuristics but the one which sorts only by goal number—is that the categories of goals in the second sorting step are very fine-grained. This leads to situations where, for example, all disjunction goals are always chosen before all high priority premise goals, regardless of their respective ages. The problem with this can be readily seen when a lemma is written which introduces a disjunction goal, such that solving this goal results in a case where the lemma can be applied again—and thus introduces another disjunction goal to solve, and so on. The '`s`' heuristic is almost certainly incapable of proving any such lemma, as the only goals that would ever be solved in this process are deconstruction chain goals and the repeatedly introduced disjunction goals, which are both sorted so that they are solved before any other goal category. In Section 6.1.4 we construct a new heuristic focused on analysis of stateful protocols; our heuristic greatly reduces the chances of ending up in a non-terminating loop, and adjusts the ordering of several categories of goals to be more suitable for stateful protocols.

6.1.2 Heuristic locality and annotations

Notably, a rule with an open premise goal for a fact Fa would be sorted identically to a premise goal for Fa created by any other rule. The heuristics support a method for users to manually provide information about solving priority, by prioritizing facts based on their name: goals involving facts with a name beginning in '`F_`' are solved earlier, while goals with fact names beginning in '`L_`' are delayed. But this mechanism requires changing the fact name everywhere it occurs in the model, since two fact instances with different names cannot unify; this is often too coarse, particularly in stateful models with loops.

To see why, consider a common construction in stateful protocols, where some fact is used to represent the state of an agent and there may be several rules which have that fact in both their premise and conclusion. A simple example would be the following protocol:

$$\left\{ \frac{}{\mathsf{St}(\texttt{'0'})}[\mathsf{Start}()],\ \frac{\mathsf{St}(x)}{\mathsf{St}(\texttt{'1'})}[\mathsf{Rule1}()],\ \frac{\mathsf{St}(\texttt{'1'})}{\mathsf{St}(\texttt{'2'})}[\mathsf{Rule2}()] \right\}.$$

Solving a premise goal for the St fact from the rule with action Rule2() gives only a single possible source, the rule with action Rule1(). Solving the same premise goal for St from the rule with action Rule1(), however, gives a case distinction with three different options. In real protocols, this is quite common: receiving a second message likely depends on the mode that was set in, and fresh values that were generated in, a particular prior rule. As such, solving a premise goal for the source of the current state fact may be quite useful when there are strict constraints on which rules may have been executed just prior. On the other hand, an initial rule may be executable from a great many different states, and so solving for that premise would introduce a case split over all such options.

This problem was identified early in Tamarin's development, and lead to a precomputation step to annotate so-called *loop breakers*, which were solved with lower priority to prevent looping behaviour (for information about how these are computed and marked in Tamarin, see [63, Page 143]). This approach works well in early case studies for Tamarin, but the results are only occasionally useful for more complex looping protocols. Though it is valuable to have different solving priorities for different premise goals of the same fact depending on where they occur, the loop breakers are an entirely automated precomputation step with no way of manually influencing the behaviour. Their unpredictable nature and tendency to interfere in larger protocols means that Tamarin's heuristics have a mode to ignore loop breakers, prioritizing them normally instead of delaying them. This mode can typically be chosen by capitalizing the name of the heuristic, e.g. '`S`' instead of '`s`'; note that we will do this for all of our case studies in Chapter 7.

We resolve this problem in a different way by introducing a general approach to annotating rules and facts, and extending Tamarin's syntax so that annotations can be expressed in protocol models. These annotations capture several disparate implementations needed for particular features added to Tamarin over time, including loop breakers and *diff* rule annotations. The former was previously stored per-rule as an array of their indices in the rule premises, and the latter as actions with syntactically-incorrect names to avoid name conflicts, which were explicitly filtered out when rules were displayed in the interface.

In addition, syntactic expression of annotations allows us to implement some useful features, ranging from the merely cosmetic annotation of rules with a colour for rendering graphs, to heuristic annotations which allow a modeller to manually

influence the priority of solving goals from particular rules.

Importantly, annotations do not influence the semantics of a rule or fact. They are ignored when checking whether two facts are unifiable, and thus a fact that has some particular annotations can unify to one without them (or to one with different annotations). This is crucial for allowing features like annotations containing localized heuristic information; a modeller can specify information locally without concern that it might invalidate their model.

6.1.3 Syntax description

To add support for rule and fact annotations within Tamarin's syntax, we take inspiration from the existing syntax for specifying lemma attributes. There are four lemma attributes included in Tamarin:

1. `use_induction`, which ensures Tamarin transforms the lemma formula with the trace induction transformation we describe later in Section 6.2.1.
2. `reuse` which adds the formula in the lemma as a constraint for all later lemmas, allowing the property to be used to help prove later lemmas,
3. `sources` which adds the formula in the lemma as a constraint during precomputation, so that all precomputed sources will satisfy the formula, and
4. `hide_lemma=<name>` which removes the lemma of that name from the constraint system (e.g. to ignore particular prior `reuse` lemmas).

We will discuss `sources` lemmas further in Section 6.2.2. Reusable lemmas are often necessary to prove small properties that may apply to many different cases when proving a larger lemma. This can avoid Tamarin repeating nearly identical searches across an exponential number of cases. In Chapter 7, we will show many examples of `reuse` lemmas used to prove minor properties of a model. Sometimes a `reuse` lemma used to prove a larger lemma is irrelevant to the proof of later lemmas, and in these cases `hide_lemma` allows a modeller to exclude the additional irrelevant constraint where appropriate.

These lemma attributes are expressed in square brackets after the lemma identifier, before the colon separating the identifier from the definition. For example, a lemma named 'example_lemma' with the `use_induction` attribute would be expressed as follows.

```
lemma example_lemma[use_induction]:
```

Rules in TAMARIN are specified in terms of a grammar, which includes a rule identifier, an optional *letblock* binding identifiers within the rule, and the premise and conclusion facts with optional action facts between them. For example, the rule $[\,\mathsf{L}(x)\,]\!-\![\,\mathsf{A}(x)\,]\!\rightarrow\![\,\mathsf{R}(x)\,]$ named 'example_rule' would be expressed as follows.

```
rule example_rule:  [ L(x) ] -[ A(x) ]-> [ R(x) ]
```

We modify the grammar to allow annotations (shown in colour below) similarly to lemma attributes.

⟨*rule*⟩ ::= '`rule`' ⟨*ident*⟩ ['`[`'⟨*ruleannotes*⟩'`]`'] '`:`'
[⟨*letblock*⟩]
'`[`' ⟨*facts*⟩ '`]`' ('`->`' | '`-[`' ⟨*facts*⟩ '`]->`') '`[`' ⟨*facts*⟩ '`]`'

⟨*letblock*⟩ ::= '`let`' (⟨*ident*⟩ '`=`' ⟨*term*⟩)+ '`in`'

⟨*ruleannotes*⟩ ::= ⟨*ruleannotation*⟩ ['`,`' ⟨*ruleannotes*⟩]

Our *ruleannotation* addition to the grammar currently consists of only of `colour=` or `color=` followed by a hexadecimal colour code, to implement a rule colouring feature that allows a modeller to specify how a rule will appear in TAMARIN's graph output. An example of this is shown in Figure 6.1. This long-requested feature allows a modeller to more easily identify rules to roles or adversary action; for example, a user wishing for a key reveal rule coloured bright red to easily identify where it occurs in a trace may write a rule as follows.

```
rule key_reveal[colour=ff0000]:
                        [ !Key( k) ] -[ KeyReveal( k) ]-> [ Out( k) ]
```

Rule colouring is only one example of a use for rule annotations, but with our generic implementation it is easy to implement other features which make use of them.

Fact annotations are implemented similarly, and allow the heuristic priority of goals based on a fact to be indicated within TAMARIN's syntax. We add annotations to facts in the following way (our changes shown in colour).

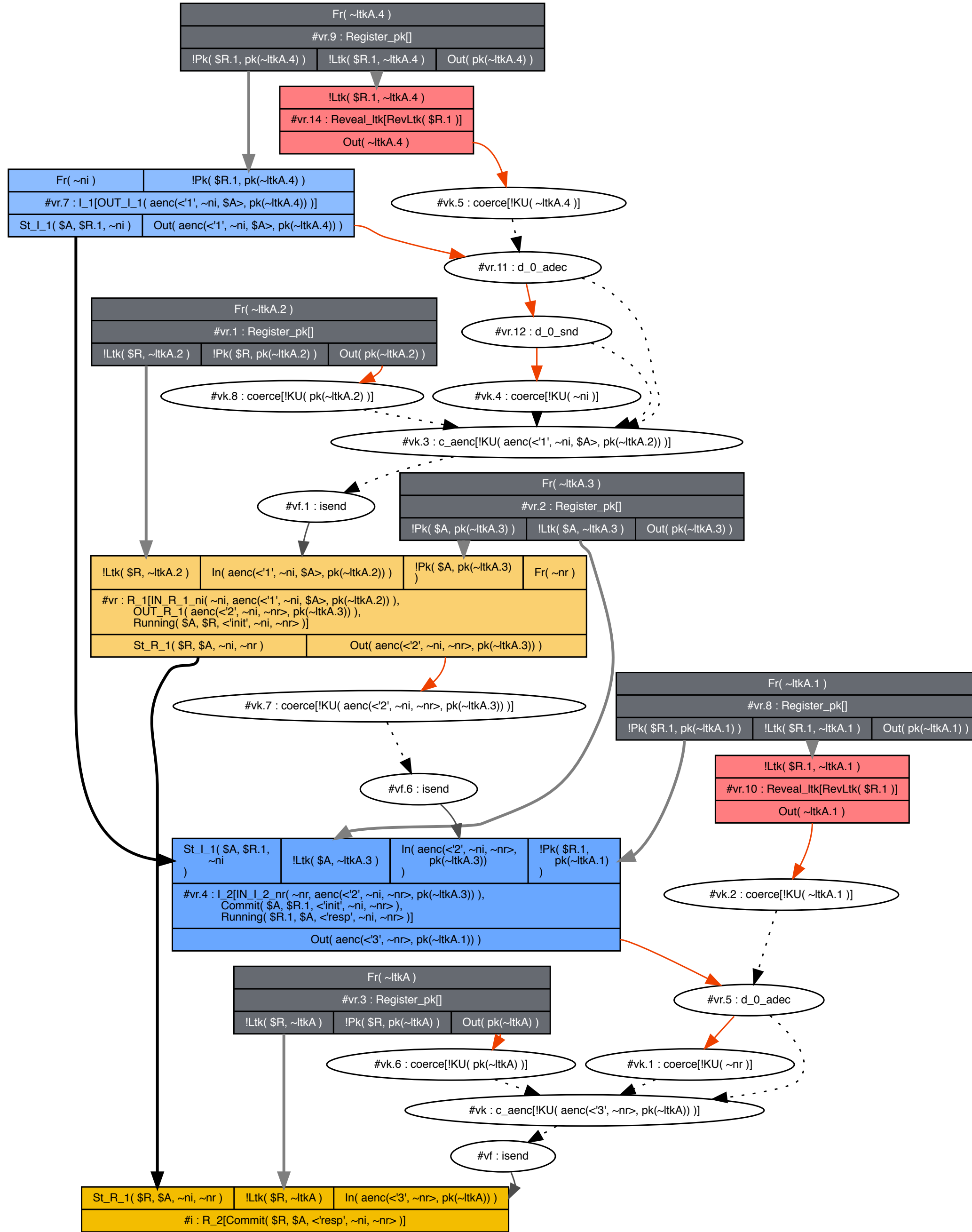

Figure 6.1: Annotations to indicate colours for rules are an example of new TAMARIN features made possible through a generic annotation interface. Here, we show an attack on the Needham-Schroeder public-key protocol [71] that TAMARIN generates, where compromise rules have been coloured red while rules executed by the roles I and R are coloured in shades of blue and gold respectively.

$\langle fact \rangle$ ::= ['!'] $\langle ident \rangle$ '(' $\langle multterms \rangle$ ')' ['['$\langle factannotes \rangle$']']

$\langle factannotes \rangle$ ::= $\langle factannotation \rangle$ [',' $\langle factannotes \rangle$]

Fact annotations are implemented in such a way that they can be easily extended for future features, but in our work we add two specific annotations to *factannotation* in the grammar above, '+' and '-', indicating higher and lower priority to the heuristic respectively. These have been implemented within all existing heuristics such that the two annotations have identical heuristic priorities to 'F_' and 'L_' respectively.

This heuristic information can be marked independently for each occurrence of a fact in a rule, and will be propagated to the instantiated facts that are created from rule instantiation. For example, in the looping protocol above, we can express the desired solving priorities in the protocol with the equivalent of

$$\left\{ \frac{}{\mathsf{St}(\texttt{'0'})}[\mathsf{Start}()],\ \frac{\mathsf{St}(x)[-]}{\mathsf{St}(\texttt{'1'})}[\mathsf{Rule1}()],\ \frac{\mathsf{St}(\texttt{'1'})[+]}{\mathsf{St}(\texttt{'2'})}[\mathsf{Rule2}()] \right\},$$

to indicate that instances of the final rule should have the associated $\mathsf{St}(\texttt{'1'})$ premise fact goal solved sooner than the heuristic would normally, while the premise goals created by the middle rule should be solved later than normal. Though this is a simple toy example, this illustrates how our changes allow a modeller can manually influence heuristic priority to achieve a desired outcome. In this case, solving the premise for the rule labelled $\mathsf{Rule1}()$ was undesirable because it introduced many cases, even though solving a premise fact of the same name in the rule labelled $\mathsf{Rule2}()$ would not introduce any additional cases. Previously, a modeller would have no choice but to prioritize both or neither; our work allows them to pick individually which instances should be prioritized by Tamarin.

6.1.4 Tailoring the heuristic

Our heuristic is built from the foundations of the 's' heuristic, with adjustments to suit common constructions in stateful protocols and a design change to alleviate the looping problem we observed above. We change only the second sort performed, where we define only four categories of goal instead of the twelve listed above.

1. Premise and action goals marked as 'immediate' priority, including

   (a) premise and action goals for facts which have names beginning in `I_`,
   (b) action goals for adversary deduction of a term x such that the name of x begins with `I_`.

2. High priority goals, including
   (a) premise and action goals for facts annotated with `[+]` or which have names beginning in `F_`,
   (b) deconstruction chain goals,
   (c) action goals for adversary deduction of a term x such that $x \in \mathcal{V}_{fresh}$, or the name of x begins with `F_`.

3. Medium priority goals, including
   (a) action goals that are not adversary deduction actions,
   (b) disjunction goals,
   (c) action goals for adversary deduction of a term with exactly one precomputed source, or where the term is a private function,
   (d) premise goals,
   (e) deferred case splits that result in less than three cases.

4. Low priority goals, which are
   (a) action goals for adversary deduction of a term x where either $\exists m, k \, . \, x = \mathtt{sign}(m, k)$, or $\exists g, y_1, y_2 \, . \, x = g^{y_1 * y_2}$.

Other than the category structure, this heuristic has a few notable differences in priority compared to the '`s`' heuristic above. First, a new 'immediate' priority is added, so that premise and action goals can be marked and solved before any other goal. This can be useful for forcing particular goals to be solved immediately, which causes the constraint solver to act similarly to how it would if that goal were solved as part of the precomputation of sources. This can be useful, for example, when there is a known invariant that should be solved but a `sources` lemma is difficult to prove.

Along with the addition of the immediate priority goals, a few categories of goals have their priorities adjusted. Goals for adversary deduction actions of a term $x \in \mathcal{V}_{fresh}$ are made higher priority, since they typically introduce ordering constraints necessary for TAMARIN to discover contradictions of injectivity. Disjunction goals are

now prioritized the same as many other categories of goals, rather than being above even the goals annotated to be high priority. This reflects the need to express more case distinctions in helper lemmas (e.g. for disjunctions over the possible sources for a loop invariant), compared to protocols without loops where the relevant disjunctions are already introduced through the precomputed sources.

The resulting heuristic performs substantially better on stateful protocol models. All the case studies we present in Chapter 7 make use of this heuristic, and excepting the timings presented in the WireGuard benchmarks, none terminate in under an hour when run with the original heuristic.

On non-stateful examples, our heuristic generally offers little benefit. When run over 84 protocols from TAMARIN's included examples, analysis time was roughly the same on average but varied by protocol[1]. It is important to note that there is significant selection bias in this sample: only examples which terminate in reasonable time under the existing heuristic are included as TAMARIN example files. As such, certain pathological cases written with the smart heuristic in mind perform poorly.

The most common source of issues seen when running early examples under the new heuristic is in cases where a model generates partial deconstructions (i.e. when not all sources of message terms can be precomputed). When this occurs, prioritizing adversary deduction of fresh term is detrimental, as all partial deconstructions can be taken as potential sources of that term. This can be resolved through the use of `sources` lemmas, which we will discuss in the next section.

6.2 Loop invariants

One limitation of the original constraint solving algorithm used by TAMARIN is that a protocol model containing loops (i. e. some rule or sequence of rules may be applied repeatedly) can cause non-termination [81]. To solve this, Meier [63] extends the algorithm with trace induction. We begin by reviewing the motivation for TAMARIN's trace induction and its implementation.

The intuition behind trace induction is to identify an invariant in a loop, and use it to 'skip over' repeated executions of that loop to consider properties of the execution prior to it. For example, one might have a looping protocol in which certain terms like long-term keys are not modified. To prove that, e.g. the long-term key

[1] An average reduction in analysis time of 0.56% with a standard deviation of 20.7%. The maximum reduction seen was 47.5%.

was generated in a particular rule, trace induction works by proving that each time the long-term key appears in some action in a trace, it is either preceded by the same action containing the same long-term key earlier in the trace (and thus that the long-term key is invariant), or preceded by the rule in which it was generated. Without trace induction, such a proof would not be possible: performing constraint reduction leads to constraint systems in which the loop is executed an unbounded number of times before any preceding rules.

Despite the usefulness of trace induction, it has several limitations in the context of analyzing stateful protocols. These limitations point us to a modelling technique which makes invariants explicit within a model, though doing so requires changing the semantics of a model. We describe this technique in Section 6.2.2; by examining its efficacy, we see how a modification to TAMARIN's precomputation allows us to achieve similar results using so-called `sources` lemmas. Our contribution to improving TAMARIN's precomputation allows for formerly-intractable models to be solved efficiently, as we will show in Chapter 7. Finally, in Section 6.2.3 we address another limitation of trace induction—the introduction of redundant action goals—with a modelling technique. Though this technique is easy to apply, we hope that it will inspire further development to TAMARIN that will make it unnecessary.

6.2.1 Trace induction and limitations

Meier gives a canonical example which motivates trace induction in [63], which we reproduce below.

Example 26. Consider the protocol

$$P_{Looping} = \left\{ \frac{\mathsf{Fr}(x)}{\mathsf{A}(x)}[\mathsf{Start}(x)],\ \frac{\mathsf{A}(x)}{\mathsf{A}(x)}[\mathsf{Loop}(x)] \right\}.$$

Using the constraint solving algorithm given in [81] to prove the trace property

$$\varphi = \forall x, i \ .\ \mathsf{Loop}(x)\ @\ i \Rightarrow \exists j\ .\ \mathsf{Start}(x)\ @\ j, \tag{6.1}$$

we must show that there are no solutions to the corresponding constraint system

$$\Gamma = \{\exists x, i\ .\ \mathsf{Loop}(x)\ @\ i \wedge (\forall j\ .\ \mathsf{Start}(x)\ @\ j \Rightarrow \bot)\}\,.$$

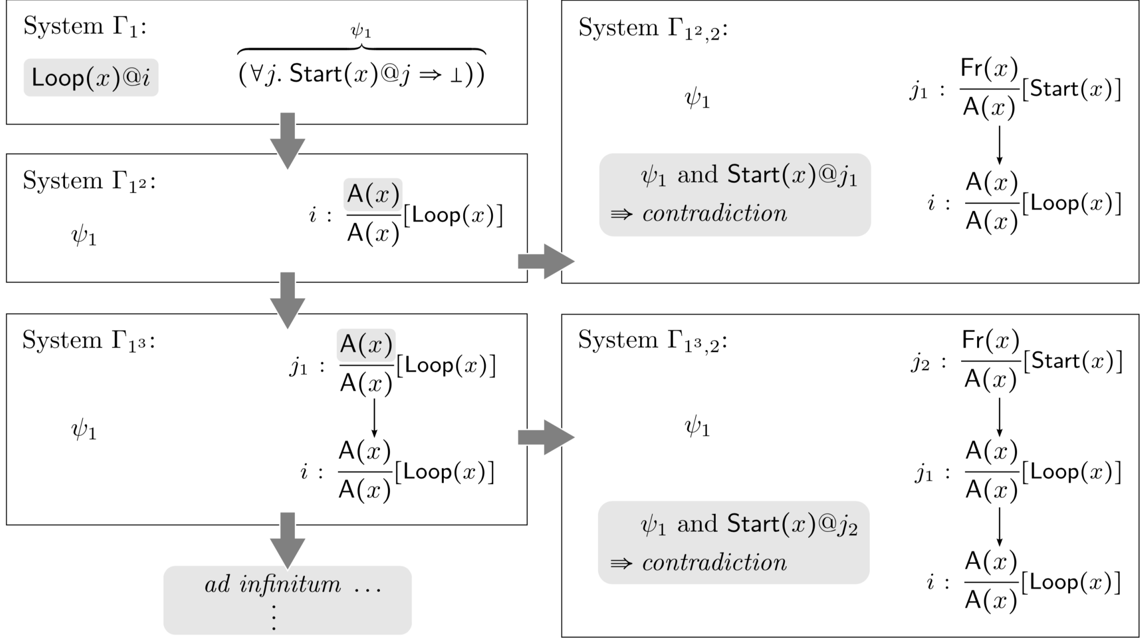

Figure 6.2: Figure from [63] depicting the constraint systems and constraint reduction steps in Example 26

Applying constraint-reduction steps to Γ gives the constraint systems shown in Figure 6.2. Repeatedly applying the reduction step $\mathtt{DG}_{\blacktriangleright}$ to add the new premise constraint for $\mathsf{A}(x)$ and solving it with $\mathtt{S}_{\blacktriangleright}$ (as we did in Example 25) always leaves one consistent constraint system, and no other reduction rules are applicable—we cannot prove φ.

Though the property above seems obvious, it cannot be proven with the original constraint solving algorithm. To tackle this problem, Meier [63] formalizes a notion of trace induction, comprising

1. a way to represent the last timepoint in a trace, introducing a trace atom and enforcing that it occurs last in every trace with extensions to the existing satisfaction relation and constraint-reduction relation;

2. a transformation of trace formulae which (informally) restricts the set of satisfying traces to 'minimal' ones, in the sense that the satisfying traces after the transformation are those which satisfy the original formula and have no prefix that does.

The trace atom, written $\text{last}(i)$ for a timepoint i, represents that i is instantiated with the last index of the trace. The satisfaction relation $\vDash_E$ is extended to ensure

$$\mathsf{S}_{\text{last},\lessdot}: \quad \Gamma \rightsquigarrow \bot \qquad \text{if } (\text{last}(i)) \in \Gamma \wedge (\exists j : ri \,.\, j \in \Gamma \wedge i \lessdot_\Gamma j)$$

$$\mathsf{S}_{\text{last},\text{last}}: \quad \Gamma \rightsquigarrow (i \doteq j, \Gamma) \qquad \text{if } \{\text{last}(i), \text{last}(j)\} \subseteq \Gamma; \boxed{i \neq j}$$

$$\mathsf{S}_{\neg,\text{last}}: \quad \Gamma \rightsquigarrow (i \lessdot j, \text{last}(j), \Gamma) \parallel (\text{last}(j), j \lessdot i, \Gamma)$$
$$\text{if } \text{last}(i) \notin \Gamma \text{ and } j \text{ fresh}; \boxed{\neg (\exists k \,.\, \text{last}(k) \in \Gamma \wedge (i \lessdot_\Gamma k \vee k \lessdot_\Gamma i))}$$

Figure 6.3: Constraint reduction rules added for the last trace atom.

that this timepoint is last, i.e. for every trace tr, valuation θ, and equational theory E,

$$(tr, \theta) \vDash_E \text{last}(i) \iff \theta(i) = |tr|.$$

Additionally, the constraint reduction relation $\rightsquigarrow_{R,E}$ is extended with constraint reduction rules which ensure that last timepoints are unique and are not less than any other and the constraint reduction rules are shown in Figure 6.3

For trace formulas without last atoms (called *last-free* trace formulas), two trace formula transformations are defined, BC() and IH(), representing the base case and inductive hypothesis respectively. These transformations are shown in Figure 6.4, and Meier proves the following theorem (the proof can be found in [63, Page 184]).

Theorem 27 (Trace Induction)**.** *Let T be a prefix closed set of traces. For every closed, last-free trace formula φ, it holds that*

$$T \vDash^{\forall}_{E} \varphi \iff T \vDash^{\forall}_{E} \textsc{bc}(\varphi) \wedge (\textsc{ih}(\varphi) \Rightarrow \varphi).$$

Theorem 27 implies a transformation we can apply to last-free trace formulas when searching for satisfying traces of a protocol. Specifically, because satisfiability and validity are dual, Theorem 27 admits the following corollary.

Corollary 28. *For a prefix closed trace set T and a last-free trace formula φ,*

$$T \vDash^{\exists}_{E} \varphi \iff T \vDash^{\exists}_{E} \textsc{bc}(\varphi) \vee (\neg\textsc{ih}(\varphi) \wedge \varphi).$$

Informally, this represents that, if there is a satisfying trace in T, then we should be able to find a 'minimal' satisfying trace—none of its prefixes is satisfying.

Revisiting Example 26, we can now transform the formula φ in Equation 6.1 to

$$\textsc{bc}(\varphi) := \begin{cases} \bot & \text{if } \varphi = f \mathbin{@} i \\ \neg\textsc{bc}(\varphi_1) & \text{if } \varphi = \neg\varphi_1 \\ \textsc{bc}(\varphi_1) \wedge \textsc{bc}(\varphi_2) & \text{if } \varphi = \varphi_1 \wedge \varphi_2 \\ \exists x \,.\, \textsc{bc}(\varphi_1) & \text{if } \varphi = \exists x \,.\, \varphi_1 \\ \varphi & \text{if } \varphi \text{ is a trace atom other than } f \mathbin{@} i \end{cases}$$

$$\textsc{ih}(\varphi) := \begin{cases} \neg\textsc{ih}(\varphi_1) & \text{if } \varphi = \neg\varphi_1 \\ \textsc{ih}(\varphi_1) \wedge \textsc{ih}(\varphi_2) & \text{if } \varphi = \varphi_1 \wedge \varphi_2 \\ \exists i \,.\, \textsc{ih}(\varphi_1) \wedge \neg\mathrm{last}(i) & \text{if } \varphi = \exists i : \mathit{temp} \,.\, \varphi_1 \\ \exists x \,.\, \textsc{ih}(\varphi_1) & \text{if } \varphi = \exists x : s \,.\, \varphi_1 \text{ and } s \text{ is a subsort of } \mathit{msg} \\ \varphi & \text{if } \varphi \text{ is a trace atom} \end{cases}$$

Figure 6.4: Meier's trace property transformations $\textsc{bc}(\varphi)$ and $\textsc{ih}(\varphi)$.

the formula $\varphi' = \textsc{bc}(\varphi) \wedge (\textsc{ih}(\varphi) \Rightarrow \varphi)$, i.e.

$$\begin{aligned} \varphi' = {} & (\forall x, i \,.\, \bot \Rightarrow \exists j \,.\, \bot) \wedge \\ & \Big((\forall x, i \,.\, \mathsf{Loop}(x) \mathbin{@} i \Rightarrow \mathrm{last}(i) \vee (\exists j \,.\, \mathsf{Start}(x) \mathbin{@} j \wedge \neg\mathrm{last}(j))) \\ & \qquad \Rightarrow (\forall x, i \,.\, \mathsf{Loop}(x) \mathbin{@} i \Rightarrow \exists j \,.\, \mathsf{Start}(x) \mathbin{@} j)\Big), \end{aligned}$$

such that

$$P_{\mathit{Looping}} \vDash^{\forall}_{E} \varphi \iff P_{\mathit{Looping}} \vDash^{\forall}_{E} \varphi'.$$

From the new formula φ', the constraint solving algorithm derives the corresponding constraint system characterizing all counterexamples of the above,

$$\begin{aligned} \Gamma' := \{ \; & (\exists x, i \,.\, \mathsf{Loop}(x) \mathbin{@} i \wedge (\forall j \,.\, \mathsf{Start}(x) \mathbin{@} j \Rightarrow \bot)), \\ & (\forall x, i \,.\, \mathsf{Loop}(x) \mathbin{@} i \Rightarrow \mathrm{last}(i) \vee (\exists j \,.\, \mathsf{Start}(x) \mathbin{@} j \wedge \neg\mathrm{last}(j))) \; \}. \end{aligned}$$

Applying constraint reductions to Γ' creates a $\mathsf{Loop}(x)$ action at i from the first constraint, and adds the trace atom last from the case distinction in the second constraint. Now, the $\mathtt{DG}_{\blacktriangleright}$ reduction which looped indefinitely in Example 26 adds another last atom from the second constraint, and the case can be reduced to $\bot$ by applying $\mathtt{S}_{\mathrm{last},\lessdot}$. The transformed trace formula has been proven with our modified constraint reduction algorithm.

The example above shows the power of trace induction, and it is important to prove

nearly any property of a protocol with looping behaviour. Nonetheless, trace induction has some limitations that make it unwieldy for protocols with several interacting loops, as found in many stateful protocols. For example, with more complex looping behaviour, proving a loop invariant through trace induction may require building up to the final property in parts.

Example 29. Consider the protocol

$$P_{Looping'} = \left\{ \frac{\mathsf{Fr}(x)}{\mathsf{St}(x)}[\mathsf{Start}(x)],\ \frac{\mathsf{St}(x)}{\mathsf{St}(x)}[\mathsf{Loop1}(x)],\ \frac{\mathsf{St}(x)}{\mathsf{St}(x)}[\mathsf{Loop2}(x)] \right\},$$

for which we wish to prove the trace property

$$\varphi = \forall x, i \,.\, \mathsf{Loop2}(x) \ @ \ i \Rightarrow \exists j \lessdot i \,.\, \mathsf{Start}(x) \ @ \ j.$$

Applying the trace induction transformation as above, we get the constraint system

$$\Gamma = \Big\{ \Big(\exists x, i \,.\, \mathsf{Loop2}(x) \ @ \ i \wedge (\forall j \,.\, \mathsf{Start}(x) \ @ \ j \Rightarrow \neg(j \lessdot i)) \Big), \\ \Big(\forall x, i \,.\, \mathsf{Loop2}(x) \ @ \ i \Rightarrow \text{last}(i) \vee (\exists j \lessdot i \,.\, \mathsf{Start}(x) \ @ \ j \wedge \neg\text{last}(j)) \Big) \Big\}.$$

In solving this constraint system, $\mathsf{Loop2}(x)$ is instantiated by instantiating a rule at timepoint i, which must also be the last timepoint. This gives a system with an open constraint on the first premise of the rule (i.e. the constraint $\mathsf{St}(x) \blacktriangleright_1 i$). Solving this constraint by applying the relevant constraint reduction ($\mathtt{DG}_\blacktriangleright$) gives three systems, of which two are contradictory—one by the induction transformation, and one from the original constraint. The third instantiates the rule containing $\mathsf{Loop1}(x)$, leaving a similar open constraint on the $\mathsf{St}(x)$ premise. Continuing to reduce this system will simply loop, generating a similar set of subsystems each time.

To prove our trace property, we must add an additional constraint to prevent the premise case $\mathsf{Loop1}(x)$ from looping indefinitely. For example, the property

$$\psi_{\mathsf{Loop1}} = \forall x, i \,.\, \mathsf{Loop1}(x) \ @ \ i \Rightarrow (\exists j \lessdot i \,.\, \mathsf{Start}(x) \ @ \ j) \vee (\exists j \lessdot i \,.\, \mathsf{Loop2}(x) \ @ \ j)$$

proves a weaker invariant of $\mathsf{Loop1}(x)$ and can be solved easily with trace induction—it is nearly the same as the constraint system as for φ, except that all three solutions to the $\mathsf{St}(x)$ premise are contradictory. We can then 'reuse' this invariant by replacing

our property φ with a modified property

$$\varphi' = (\psi_{\mathsf{Loop1}} \Rightarrow \varphi)\,.$$

Applying the trace induction transformation to this property gives a constraint system equivalent to

$$\begin{aligned}\Gamma' = \Big\{ &\Big(\forall x, i\ .\ \mathsf{Loop1}(x)\ @\ i \Rightarrow (\exists j \lessdot i\ .\ \mathsf{Loop2}(x)\ @\ j) \vee (\exists j \lessdot i\ .\ \mathsf{Start}(x)\ @\ j)\Big), \\ &\Big(\exists x, i\ .\ \mathsf{Loop2}(x)\ @\ i \wedge (\forall j\ .\ \mathsf{Start}(x)\ @\ j \Rightarrow \neg(j \lessdot i))\Big), \\ &\Big(\Big(\exists x, i\ .\ \mathsf{Loop1}(x)\ @\ i \wedge \neg\text{last}(i) \wedge (\forall j\ .\ \mathsf{Start}(x)\ @\ j \Rightarrow \neg(j \lessdot i)) \\ &\qquad \wedge (\forall j\ .\ \mathsf{Loop2}(x)\ @\ j \Rightarrow \neg(j \lessdot i))\Big) \vee (\forall x, i\ .\ \mathsf{Loop2}(x)\ @\ i \Rightarrow \text{last}(i))\Big)\Big\}.\end{aligned}$$

In solving this, the second constraint instantiates $\mathsf{Loop2}(x)$ at some timepoint i. The system is then split over the disjunction in the third constraint: the first case instantiates a $\mathsf{Loop1}(x)$, which immediately implies a contradiction by the first constraint; the second case implies last(i) and leaves an open premise constraint on $\mathsf{St}(x) \blacktriangleright_1 i$ as above. This time, however, the case containing $\mathsf{Loop1}(x)$ implies a contradiction because of the first constraint.

While it is relatively straightforward to apply trace induction to prove reusable lemmas that achieve the desired outcome in Example 29, larger protocols compound this problem. Invariant properties that have more than one possible loop would need to be built up in parts, proving weaker properties and compounding those to prove the actual property that is relevant. Worse, the weaker invariants needed to build up to a larger property are likely to slow down TAMARIN's analysis by introducing many unnecessary goals; for example, in Section 7.2.2 we see a case study where proving invariants through induction lemmas without modifying TAMARIN would result in every rule instance introducing at least four separate action goals.

The difficulty of building up invariant properties and the relative inefficiency of solving the implied goals has practical effects on previous protocol analysis work in TAMARIN; the analysis of TLS 1.3 by Cremers *et al.* [28] required writing tens of reusable lemmas to prove invariants, and splitting the model across multiple files to avoid unnecessary interaction between these lemmas for performance reasons. Despite this, analysis of the model still required several days of computation on a modern server.

To counter this, we begin by examining how TAMARIN deals with explicit invariant constraints added to rule premises, a modelling technique we developed that dramatically improves performance compared to equivalent reusable lemmas. This technique requires the modeller to manually prove that the invariants exist—i.e., that including them in the rule premises is valid—a significant downside for automated analysis. By determining why this technique is so effective, however, we develop improvements to TAMARIN that allow similar performance to be achieved without this proof burden, as we shown by benchmarks in Section 7.2.1. Furthermore, we develop a method for proving invariants inductively without introducing irrelevant goals when solving other lemmas.

6.2.2 Improving precomputation for handling invariants

When dealing with invariants in a protocol model, reusable lemmas that establish the progenitor of a particular invariant—that is, the constraints on actions introduced by reusable lemmas—are notably less efficient than premise constraints. In part, this is because of how the solver heuristic determines the order to apply constraint reductions, especially those involving disjunctions—an issue we discussed in Section 6.1. The majority of this difference, however, can be found in the differing ways premises and actions interact with the precomputation done by TAMARIN.

Recall from Section 5.5 that TAMARIN performs precomputation on a model in order to compute 'sources' of premise goals and adversary deduction goals, where these sources are precomputed systems solving as many implied goals as possible without introducing further case splits. When solving a premise goal in a constraint system, TAMARIN first determines whether there is a precomputed source for that premise goal, and if so, unifies the precomputed system directly. If a goal does not have a precomputed source then the additional implied goals must be solved one by one, even if they only have trivial solutions; thus, depending on the model, goals without precomputed sources can be substantially slower than those that do.

Here, we will briefly describe a modelling technique which can be used to dramatically improve TAMARIN's performance on looping protocols, taking advantage of the way precomputation of premise goals works. This technique comes at the cost of the manual proof burden to show that changing a model's semantics is sound, and we discuss why the existing TAMARIN framework prevents us from achieving similar results without changing the semantics of a model. We remedy this by modifying

Tamarin's precomputation step to apply precomputed sources to action goals. In combination with a special type of lemma that allows us to prove loop invariants within these precomputed sources, this allows us to have invariant properties applied directly when reasoning about protocols which results in much greater performance than previously possible.

Explicit invariant constraints

Given that premise goals are precomputed, one way we can ensure the precomputation directly considers invariants is by adding a premise constraint for the relevant invariants. Recall that Tamarin supports persistent facts—facts which are not removed from global state when used in the premise of a rule. Persistent facts are ideal for representing invariants, and by adding one into the conclusions of the rules where the invariants are set, and into the premises of all rules which use them, we can ensure that the invariants both have their sources precomputed directly and are considered as constraints in the precomputation of other sources. We give an example of a simple modified protocol here, and a full model with these explicit invariant constraints can be found in Appendix A.1.1. In Section 7.2.1, we give an example of the performance of this method compared to other models of the same protocol.

Example 30 (Explicit invariants)**.** Recall the protocol $P_{Looping}$ from Example 26. Here, we discuss a minor modification of this protocol to include an $\mathsf{End}(x)$ action, $P_{LoopEnd}$. This will allow us to separate the loop itself from the property we wish to prove later.

$$P_{LoopEnd} = \left\{ \frac{\mathsf{Fr}(x)}{\mathsf{A}(x)}[\mathsf{Start}(x)],\ \frac{\mathsf{A}(x)}{\mathsf{A}(x)}[\mathsf{Loop}(x)],\ \frac{\mathsf{A}(x)}{}[\mathsf{End}(x)] \right\}.$$

To modify the $P_{LoopEnd}$ protocol to include explicit invariants, we add a new persistent fact $!\mathsf{I}(x)$ representing the invariant x to the conclusion of the rule labelled $\mathsf{Start}(x)$ and to the premises of the rules labelled $\mathsf{Loop}(x)$ and $\mathsf{End}(x)$ to get a new protocol,

$$P_{Invars} = \left\{ \frac{\mathsf{Fr}(x)}{\mathsf{A}(x), !\mathsf{I}(x)}[\mathsf{Start}(x)],\ \frac{\mathsf{A}(x), !\mathsf{I}(x)}{\mathsf{A}(x)}[\mathsf{Loop}(x)],\ \frac{\mathsf{A}(x), !\mathsf{I}(x)}{}[\mathsf{End}(x)] \right\}.$$

In our modified protocol, the precomputation of sources for $\mathsf{I}(x)$ begins with $!\mathsf{I}(x) \blacktriangleright_1 i$,

and results in the constraint system

$$\Big\{ \mathsf{!I}(x) \blacktriangleright_1 i, \frac{\mathsf{Fr}(x)}{\mathsf{A}(x), \mathsf{!I}(x)}[\mathsf{Start}(x)] : j, (j, 2) \rightarrowtail (i, 1) \Big\}.$$

Precomputation of sources for $\mathsf{A}(x)$ begins with $\mathsf{A}(x) \blacktriangleright_1 i$ and results in two constraint systems,

$$\Big\{ \mathsf{A}(x) \blacktriangleright_1 i, \frac{\mathsf{Fr}(x)}{\mathsf{A}(x), \mathsf{!I}(x)}[\mathsf{Start}(x)] : j, (j, 1) \rightarrowtail (i, 1) \Big\},$$

and

$$\Big\{ \mathsf{A}(x) \blacktriangleright_1 i, \frac{\mathsf{A}(x), \mathsf{!I}(x)}{\mathsf{A}(x)}[\mathsf{Loop}(x)] : j, (j, 1) \rightarrowtail (i, 1), \mathsf{!I}(x) \blacktriangleright_2 j, \frac{\mathsf{Fr}(x)}{\mathsf{A}(x), \mathsf{!I}(x)}[\mathsf{Start}(x)] : k, (k, 2) \rightarrowtail (j, 2) \Big\},$$

representing the two sources of $\mathsf{A}(x)$, the latter also solving the $\mathsf{!I}(x) \blacktriangleright_2 j$ constraint as part of the precomputation. Note that the precomputed source for $\mathsf{!I}(x)$ and both precomputed sources for $\mathsf{A}(x)$ include a rule labelled with $\mathsf{Start}(x)$.

Thus if we attempt to solve the trace property

$$\varphi = \forall x, i \,.\, \mathsf{End}(x) \ @\ i \Rightarrow \exists j \lessdot i \,.\, \mathsf{Start}(x) \ @\ j,$$

then solving either premise of the $\mathsf{End}(x)$ rule applies a precomputed sources which immediately instantiate $\mathsf{Start}(x)$ in all cases, proving the lemma with no further steps.

Unfortunately, this approach has a serious drawback for formal verification. Adding explicit invariant constraints to rules changes the semantics of the model, introducing a proof burden for the modeller that cannot be discharged by TAMARIN, unlike a reusable lemma. In Example 30 above, our addition of the invariant premises implicitly assumes that the property φ is true to begin with, and so modifying the protocol in this way requires a corresponding proof of soundness. If we wish to take advantage of precomputed sources without manual proof burden, we must consider other approaches to influencing precomputation.

Precomputation of action goals

In order to achieve equivalent performance to the technique above without manual proof burden, we implement precomputation of action goals within TAMARIN. To

see why this is necessary, we first examine why the existing precomputation is not sufficient. We'll then discuss our modifications to the precomputation step, and give a brief example of how it influences the proof steps of a simple protocol property.

During the development of TAMARIN, it became clear that in some cases it would be necessary to specify properties that are assumed true during precomputation, to be proven later as lemmas. This was done through a special type of lemma, annotated as a '`sources`' lemma, which is assumed true in the process of precomputing systems and can be later proven in TAMARIN itself (without making use of any precomputation that may have assumed it).

Originally, this capability was used to express so-called 'type assertions' about message variables—characterizations of all the ways those message variables might be instantiated (for more about why this is useful and how this is applied, see [63, Section 8.4.4]). Note that in earlier references, `sources` lemmas may be referred to as '`typing`' lemmas for this reason.

Given that `sources` lemmas influence precomputation without introducing manual proof burden, it would seem at first glance that they are ideal for our purposes. It's straightforward to test whether this is the case; we can change an existing `reuse` invariants lemma to be a `sources` lemma. As we'll see in Section 7.2.1, however, this can prevent the proof from terminating entirely.

We can resolve this by marking the lemma both `reuse` and `sources`, which indeed shows a substantial speed up over a simple `reuse` lemma in that case study. But why is it that a `sources` lemma is not sufficient?

The primary limitation of the existing precomputation is that only premise goals and special adversary deduction goals are precomputed—specifically, action goals in protocol rules are not. Thus, precomputed sources are only used when solving premise goals, and not when solving action goals. Thus, even with the aid of a `sources` lemma, it is possible to instantiate the action referenced in the `sources` lemma in a constraint system where the lemma is not solved, whenever the rule which includes that action is instantiated through an action goal instead of a premise goal. In fact, this happens frequently: all constraint systems created from lemmas necessarily begin by solving actions.

If the `sources` lemma is marked as reusable, the lemma is added to the constraint system explicitly, and so TAMARIN can solve the constraint as an additional proof step. This is much slower than directly unifying a source system, particularly when

the source system includes many other goals solved during precomputation; we will show in our benchmarks of comparable WireGuard models in Section 7.2.1 that this can considerably slow down even a protocol with only one loop. This implies that `sources` lemmas could be substantially more effective than they are in their existing implementation.

To improve on this limitation, we modify how action goals are solved within Tamarin so that they may make use of precomputed sources. Recall from Section 5.5 that prior to our modifications precomputation in Tamarin was performed over two types of goals:

1. premise goals, generated automatically based on all facts that occur in rule premises and conclusions that are not In(), Fr(), or Out() facts,

2. and adversary deduction actions for terms that are non-private functions, as well as fresh terms.

For each of these, the goal is instantiated in a proof system, and Tamarin attempts to solve all introduced goals that do not introduce further case splits, as well as apply any previously precomputed sources (each at most once) if the associated goal is left open in the resulting system.

For our modified precomputation, we generate sources for action goals in addition to the two categories above. This requires changing one constraint applied during the graph simplification process, the `c5` step that immediately instantiates 'unique' actions—actions that occur in only one rule. Fortunately, this step is an optimization added during the implementation of Tamarin as a way of eagerly applying the `S @` reduction represented by action goals, and does not replace it.

This change also requires a modification to how action goals are solved *during* precomputation of a particular source. Previously, action goals were solved eagerly during precomputation, which commonly introduces infinite loops when `sources` lemmas are used. If solving an action goal introduced another of the same action goal, then the precomputation would loop indefinitely solving these goals. With our modified precomputation, action goals now have valid precomputed systems associated with them, and so rather than solving them eagerly during the precomputation step we can rely on the step that saturates sources with respect to each other to solve them in other systems. In this step, precomputed actions with only one possible source are solved directly, and others remain as unsolved goals in the system. Infinite loops

are prevented by the existing mechanism to limit the number of times saturation is applied.

Precomputing actions provides moderately faster analysis of existing protocols at the cost of additional precomputation time when loading theories, but the real strength of this modification is in how it modifies the way sources lemmas may be used. Since action goals are now solved with precomputed systems, we can be sure that these invariants will apply correctly in every system that gets instantiated, as we will see in the case study in Section 7.2.1. And since `sources` lemmas no longer create infinite loops if they instantiate an action referenced in another `sources` lemma, we can use them to prove invariants over much more complex protocols, as we will see in Section 7.2.2. Below, we give an example of how precomputation of actions allows `sources` lemmas to immediately apply invariants.

Example 31 (Invariants in precomputed sources)**.** Recall the protocol $P_{LoopEnd}$ from Example 30,

$$P_{LoopEnd} = \left\{ \frac{\mathsf{Fr}(x)}{\mathsf{A}(x)}[\mathsf{Start}(x)],\ \frac{\mathsf{A}(x)}{\mathsf{A}(x)}[\mathsf{Loop}(x)],\ \frac{\mathsf{A}(x)}{}[\mathsf{End}(x)] \right\}.$$

Instead of modifying this protocol to include invariants in the premises and conclusion as we did in the previous example, we instead label the rules with actions containing the loop invariant x, as follows:

$$P_{sources} = \left\{ \frac{\mathsf{Fr}(x)}{\mathsf{A}(x)}[\mathsf{Start}(x), \mathsf{Source}(x)], \right.$$
$$\left. \frac{\mathsf{A}(x)}{\mathsf{A}(x)}[\mathsf{Loop}(x), \mathsf{Invar}(x)],\ \frac{\mathsf{A}(x)}{}[\mathsf{End}(x), \mathsf{Invar}(x)] \right\}.$$

We then prove the following property of $P_{sources}$ in a `sources` lemma, using trace induction technique as in Section 6.2.1

$$\varphi_{invars} = \forall x, i\ .\ \mathsf{Invar}(x)\ @\ i \Rightarrow \exists j \lessdot i\ .\ \mathsf{Source}(x)\ @\ j.$$

With this sources lemma, and our modified precomputation, the $\mathsf{End}(x)$ action has a precomputed source of

$$\left\{ \frac{\mathsf{A}(x)}{}[\mathsf{End}(x), \mathsf{Invar}(x)] : i,\ \frac{\mathsf{Fr}(x)}{\mathsf{A}(x)}[\mathsf{Source}(x)] : j, j \lessdot j \right\}.$$

Thus, if we attempt to solve the trace property

$$\varphi = \forall x, i \,.\, \mathsf{End}(x) \;@\; i \Rightarrow \exists j \lessdot i \,.\, \mathsf{Start}(x) \;@\; j,$$

then instantiating the $\mathsf{End}(x)$ action adds the precomputed source above, which immediately satisfies the property φ with no further proof steps.

Although this is a simple example, note that φ cannot be proven with trace induction without φ_{invars}, as the $\mathsf{End}()$ action need only occur once in an unbounded trace. The property φ can, however, be solved using φ_{invars} without precomputed actions, it simply takes more proof steps; in Section 7.2.1 we will show how these additional steps add up to substantial performance differences even in a protocol with a single loop.

Careful consideration of why particular modelling techniques are so effective can be very instructive for improving tools. In this case, by comparing models with explicit invariant constraints in rule premises to those with invariants captured through lemmas, we found a much more substantial underlying reason for poor performance in several cases. We developed improvements to TAMARIN's precomputation, and with these it is now possible to express invariant constraints entirely through a lemma proven within TAMARIN, with equivalent performance.

6.2.3 Reducing irrelevant goals from induction

Recall that for induction goals, there must be at least one instance of the antecedent in every nested loop, and so proving a loop invariant requires at least one instance of the associated action in every possible looping rule (or set of rules which can loop). But proving a typical invariant lemma of the form

$$\forall inv, t_1 \,.\, \texttt{Invariants}(inv) \;@\; t_1 \Rightarrow \exists t_0 \,.\, (t_0 < t_1) \wedge \texttt{InvariantSource}(inv) \;@\; t_0,$$

means that each of these actions will create a goal for the associated $\mathsf{InvariantSource}(inv)$ action. As such, a lemma of this form will create a large number of goals, even when the invariant in the rule is included only by necessity to prove the inductive lemma rather than because the terms in the invariant have any relevance to the rule itself.

It is possible to reduce the number of open action goals somewhat by associating

a fresh term to each invariant, and adding a 'uniqueness' lemma of the form

$$\forall id, inv, t_0, t_1 \ . \ \texttt{InvariantSource}(id, inv) \ @ \ t_0 \wedge \texttt{InvariantSource}(id, inv) \ @ \ t_1 \Rightarrow t_0 = t_1.$$

Still, the irrelevant action goals exist at least once, even when solving them should be deferred until they are relevant. Additionally, this creates a large number of irrelevant equality checks if the InvariantSource() goals do not need to be solved.

We examined multiple ways of resolving this, including several possible modifications to TAMARIN. One option was to introduce a new fact annotation for actions, such that constraints associated with them only apply during verification of sources lemmas. This solution would be easy to implement, and does not affect the soundness of 'all-traces' lemmas (as these prove that no trace can satisfy particular constraints, and the modification would only remove constraints from the proof system). But the same cannot be said for 'exists-trace' lemmas, or for attacks found when verifying an 'all-traces' lemma, which would mean great care must be taken when interpreting a complete trace if using this annotation anywhere in a model—an undesirable caveat for a formal verification tool.

Another possibility is to exploit the nature of heuristic fact annotations so that irrelevant goals are marked with `[-]`, and upgrading the priority by unifying with an unannotated action if one is created. Unfortunately, this is difficult to accomplish; it would require implementing variable priority annotations within lemmas, and ensuring that the unification of facts substitutes the fact annotations appropriately—both of which would require extensive changes to the way term substitutions are performed in TAMARIN.

In the end, it was easier to accomplish our objective within the existing TAMARIN framework by splitting each InvariantSource() fact into two facts with the same terms, InvariantSource1() and InvariantSource2(). We can then retain our Invariant() actions in every rule, add an additional UsingInvariant() fact to rules where the terms in the invariant are relevant to rule execution, and modify our invariant `sources` lemma to

instead prove that

$$(\forall inv, t_1 \,.\, \texttt{Invariants}(inv) \,@\, t_1 \\ \Rightarrow \exists t_0 \,.\, (t_0 < t_1) \wedge \texttt{InvariantSource1}(inv)[-] \,@\, t_0) \wedge \\ (\forall inv, t_1 \,.\, \texttt{UsingInvariants}(inv) \,@\, t_1 \\ \Rightarrow \exists t_0 \,.\, (t_0 < t_1) \wedge \texttt{InvariantSource1}(inv) \,@\, t_0),$$

and finally, add a reusable 'uniqueness' lemma of the form

$$\forall id, inv, t_0, t_1 \,.\, \texttt{InvariantSource1}(id, inv) \,@\, t_0 \wedge \texttt{InvariantSource2}(id, inv) \,@\, t_1 \\ \Rightarrow t_0 = t_1.$$

Thus, $\mathsf{UsingInvariants}()$ actions create $\mathsf{InvariantSource1}()$ goals with regular priority while regular $\mathsf{Invariant}()$ actions create them with reduced priority, and the unification of the facts in these goals is deferred until one of them is solved—which is the only way an $\mathsf{InvariantSource2}()$ action can be created. Once an $\mathsf{InvariantSource2}()$ action is created, the uniqueness lemma unifies the timepoints of all corresponding $\mathsf{InvariantSource1}()$ actions directly.

Our technique allows for complex nested loop invariants to be used much more efficiently than they would be with a naïve sources lemma, and we make use of this technique in order to perform the verification of DNP3:SAv5 described in Section 7.2.2. For an example of how this technique looks in a model with many overlapping invariants, the DNP3:SAv5 model can be found in Appendix A.2.

6.3 Concluding remarks

In this chapter, we've discussed the limitations of Tamarin's existing heuristics, trace induction, and precomputation. Our contributions improve upon these limitations in many ways. To improve upon Tamarin's heuristics, we augmented them with the ability for a modeller to specify 'local' heuristic priorities, allowing e.g. facts representing agent state to be solved with high priority in some instances and low priority in others. In addition, we designed and implemented a new heuristic more suited to stateful protocols, which also prevents loops that can occur in the original heuristics.

We built on Tamarin's precomputation to generate source for action goals, which gives `sources` lemmas much more power to improve the efficiency of Tamarin's automated proving. This change also allows us to prove loop invariants entirely within `sources` lemmas; as we will see in Section 7.2.1 and 7.2.2, this has significant consequences for proving complex stateful protocols.

Finally, we described a modelling technique to minimize one of the primary downsides of using trace induction to prove properties in looping protocols: the generation of many irrelevant action goals. These goals are created every time a new rule instance that forms a loop is created, but with the help of our heuristic annotations it is possible to ensure that solving them is deferred until the goal is relevant. Though this modelling technique requires manually labelling rules with appropriate actions, it does not change the execution of the model and does not require any manual proof of correctness. We hope that this modelling technique inspires future work to Tamarin that allows it to automatically handle these goals more efficiently.

With the improvements described in this chapter, we move to Chapter 7 to show the impact of our contributions. With the combination of an improved heuristic and improved methods of handling loop invariants, we are now able to return to the protocols from Chapter 4 and formally verify their detection properties. Further, our improvements allow us to perform automated analysis of two widely-used real-world protocols—the DNP3 Secure Authentication protocol and WireGuard—which have no prior formal analysis.

Chapter 7

Case studies

With the improvements to stateful protocol analysis in Chapter 6, we now have the tools necessary to verify more complex protocols than before. In this chapter, we begin by verifying the detection properties of the four detecting protocols described in Chapter 4. We then present the verification of two real-world protocols: WireGuard and DNP3 Secure Authentication.

All formal models are included in Appendix A. Note that these models are written in TAMARIN's syntax, which makes use of certain shorthand to express variable sorts (recall that TAMARIN makes use of four variable sorts: fr, pub, msg as discussed in Section 5.1, and *temp* as discussed in Section 5.3) . We write

- `~x` for $x : fr$,
- `$x` for $x : pub$,
- `#x` for $x : temp$, and
- `x` for $x : msg$.

Further information on TAMARIN modelling and use can be found in the TAMARIN manual [87].

7.1 Analyzing our detection protocols

In this section we discuss the formal analysis of our detection protocols described in Chapter 4. For each, we prove the soundness of their detection procedures, as well as basic agreement properties and the circumstances under which detection of misuse is guaranteed to occur. We proceed in the order they are presented in Chapter 4.

7.1.1 Formal analysis of the Counter protocol

To verify the properties of the counter protocol described in 4.2.1, we constructed a symbolic protocol model in TAMARIN. As a symbolic model, our analysis relies on standard symbolic assumptions as described in Chapter 2; we additionally assume that I and R are aware of each other's public keys through, e.g., a public key infrastructure or out-of-band communication.

In addition to the `signing` and `hashing` equational theories built into TAMARIN, we represent counter values using a successor function `S` applied repeatedly to a public term (specifically, the string `'0'`). Notably, we do not include a decrementing function that the adversary can apply to 'undo' the successor function, as the adversary can always construct any counter value themselves. Including such a function leads to non-termination during TAMARIN's analysis, because it is not currently possible to express properties of iterated functions in TAMARIN (for further discussion of this limitation, see [83]).

In our model, included in Appendix A.3.1, agents begin by generating and registering a fresh key for themselves in the `RegisterKey` rule. A registered agent can then take on a protocol role (either 'initiator' or 'responder') with another registered agent, allocating some state for carrying out that role. This state allocation is restricted so that a particular registered agent key is used to set up at most one state for each other registered key for each role. State is initialized with the counter at the fixed public value `'0'`. This state allocation is captured by the `BindState_I` and `BindState_R` rules, which allocate state for the initiator role and responder role respectively.

Once agents have allocated state with each other, the protocol rules can be triggered. In general, TAMARIN rules correspond to protocol messages, though the conditional `detect` event is expressed as two separate rules with different term restrictions. One rule is restricted to only occur with equality between the terms c_i and $S(c_{i-1})$, while the 'detect' version of the rule is restricted to inequality. We model the automatic revocation of a key through the conclusion of the 'detect' rule preventing further execution by that agent.

Though this protocol is a relatively simple stateful protocol compared to the examples we will examine later in this Chapter, it still requires some modelling techniques and reusable helper lemmas to analyze automatically in TAMARIN. We model invariants through `sources` lemmas, as described in Section 6.2, for the portions of agent state that remain invariant through protocol messages: the role, key pairing,

and state identifier. We use two separate facts for this, based on agent role, to ensure that the invariant sources are unique. We use a `sources` lemma to prove the invariants,

$$(\forall id, k_a, k_b, i \;.\; \mathsf{Invariant_I}(id, k_a, k_b) \;@\; i \Rightarrow \exists j \lessdot i \;.\; \mathsf{Source_I}(id, k_a, k_b) \;@\; j)$$
$$\wedge\; (\forall id, k_a, k_b, i \;.\; \mathsf{Invariant_R}(id, k_a, k_b) \;@\; i \Rightarrow \exists j \lessdot i \;.\; \mathsf{Source_R}(id, k_a, k_b) \;@\; j)\,.$$

We use our heuristic annotations from Section 6.1.2 to manually adjust the priority of state facts.

In addition, we prove the following reusable lemmas, which ensure that both the counter values and nonce pairings are unique. This helps to prevent the backwards search from considering parallel instances with similar session data.

$$\forall id, c, t_0, t_1 \;.\; \mathtt{Counter}(id, c) \;@\; t_0 \wedge \mathtt{Counter}(id, c) \;@\; t_1 \Rightarrow t_0 = t_1.$$

$$\forall id, ni, nr, t_0, t_1 \;.\; \mathtt{Nonces}(id, ni, nr) \;@\; t_0 \wedge \mathtt{Nonces}(id, ni, nr) \;@\; t_1 \Rightarrow t_0 = t_1.$$

We establish a strict ordering of certain events through nonce ordering and 'injectivity' lemmas, which introduce $\lessdot$ relations between certain events. The nonce ordering property orders the use of a nonce and its generation, e.g.

$$\forall r, pks, ni, nr, c, t_0, t_1 \;.\; \mathtt{session}(r, pks, \langle ni, nr, c\rangle) \;@\; t_1 \wedge \mathtt{Gen}(ni) \;@\; t_0 \Rightarrow t_0 \lessdot t_1,$$

which forces an explicit $\lessdot$ relation between the two time points. Without this lemma, such ordering would normally be introduced only when solving for adversary deduction goals; Tamarin's heuristics defer solving such goals when there is an obvious message source, marking them 'probably deducible' and only solving them if necessary to complete a trace.

The 'injectivity' lemma arises due to the relatively weak handling of injective facts in Tamarin. Injective facts are non-persistent facts which have a unique fresh identifier and are never duplicated by rules, and thus all rules that have an injective fact with the same identifier in their premises are strictly ordered with respect to each other. Because the counter mechanism relies on strict ordering constraints between multiple sessions, we need to write a helper lemma to prove that each of the rules performed by an agent is ordered with respect to the others. This requires explicitly ordering rule instances to saturate $\lessdot$ relations and slows down automatic verification.

The injectivity property we prove is

$$\forall id, n_1, n_2, n_3, n_4, n_5, t_0, t_1, t_2 \ . \ \texttt{Injectivity}(id, n_1, n_2) \ @ \ t_1 \wedge \\ \texttt{Injectivity}(id, n_2, n_3) \ @ \ t_2 \wedge \texttt{Injectivity}(id, n_4, n_5) \ @ \ t_0 \wedge (t_0 \lessdot t_2) \\ \Rightarrow \neg(t_1 \lessdot t_0),$$

where `Injectivity` actions have an identifier, a previous value, and a new value in each protocol rule.

Following these helper lemmas, we prove trace existence properties to show that valid protocol traces exist. These serve as a 'sanity check' for our model, ensuring that the protocol can be fully executed as modelled. We prove these for a single session, two consecutive sessions with the same agents, and the existence of a trace triggering detection when only one key is compromised. We prove the existence of two consecutive sessions as evidence that arbitrarily many sessions can occur between two agents, as this captures the two entry points into the main protocol loop: from the `BindState` rules and from the `m2` rules of each role. Finally, we prove the properties below.

Sound detection. If there is a `detect` event triggered for a particular key pairing, then at least one of the keys in that pairing was compromised before the `detect` event. Formally, this is stated as

$$\forall k_I, k_R, t_1 \ . \ \texttt{detect}(\mathsf{pk}(k_I), \mathsf{pk}(k_R)) \ @ \ t_1 \Rightarrow \\ (\exists t_0 \ . \ t_0 \lessdot t_1 \wedge \texttt{compromise}(k_I) \ @ \ t_0) \vee (\exists t_0 \ . \ t_0 \lessdot t_1 \wedge \texttt{compromise}(k_R) \ @ \ t_0) \, .$$

Note that this also implies soundness in traces where compromise is restricted to one of the keys.

Agreement without compromise. If no keys were compromised, then a session event by R implies a matching session event by I, with agreement on nonces and counter value. This is just a basic authentication property, ensuring that the protocol is correct in the uncompromised setting. Since counter values are unique, this also

implies injective agreement.

$$\forall k_I, k_R, data, t_1\ .\ \texttt{session}(\text{'R'}, \langle \mathsf{pk}(k_I), \mathsf{pk}(k_R)\rangle, data)\ @\ t_1 \wedge \neg\, (\exists k, t_c\ .\ \texttt{compromise}(k)\ @\ t_c) \Rightarrow \exists t_0\ .\ t_0 \lessdot t_1 \wedge \texttt{session}(\text{'I'}, \langle \mathsf{pk}(k_I), \mathsf{pk}(k_R)\rangle, data)\ @\ t_0.$$

Guaranteed detection of misuse of the initiator's key in prior sessions. Assuming k_R cannot be compromised, if I completes a session at t_3, then either every prior session completed by R has a corresponding session completed by I, or I will detect misuse of k_I. Formally,

$$\forall pk_I, pk_R, data_1, data_2, t_1, t_2, t_3,\ .\ t_1 \lessdot t_2 \wedge \texttt{begin}(\text{'R'}, \langle pk_I, pk_R\rangle, data_2)\ @\ t_2 \wedge \texttt{session}(\text{'I'}, \langle pk_I, pk_R\rangle, data_2)\ @\ t_3 \wedge \texttt{session}(\text{'R'}, \langle pk_I, pk_R\rangle, data_1)\ @\ t_1 \wedge \neg((\exists t_c, k_I\ .\ pk_R = \mathsf{pk}(k_R) \wedge \texttt{compromise}(k_R)\ @\ t_c) \Rightarrow (\exists t_0\ .\ \texttt{session}(\text{'I'}, \langle pk_I, pk_R\rangle, data_1)\ @\ t_0) \vee (\texttt{detect}(pk_I, pk_R)\ @\ t_3)\,.$$

With our modelling techniques, and using our heuristic improvements, the final model takes approximately 45 seconds to verify on a four-core Intel i7-6700K processor, using the '`I`' heuristic with our changes to precomputation from Chapter 6.

Together, these properties show that it is possible to instantiate useful detection storing only a counter value at each agent. This counter-based protocol provides a basis for modifying existing protocols achieving injective agreement to also detect secret misuse, with minimal overhead. We now examine one such case, that of Keyless SSL.

7.1.2 Formal analysis of modified Keyless SSL

The formal model of our modified Keyless SSL protocol can be found in Appendix A.3.2. It is similar in construction to the model of our example protocol discussed in Section 7.1.1, with a couple of notable differences. In the modified protocol, we embed the counter protocol in the handling of the second and third messages, but unlike the counter-based protocol a session is only considered completed after an additional message. This means that the protocol model requires three rules to model the C

role instead of the single one necessary for the analogous R role of the counter-based example.

Our model is significantly simplified compared to a real TLS handshake, containing only relevant parts of the TLS mutually-authenticated key exchange, rather than modelling the complete protocol. Messages in our model include only the terms shown in Figure 4.3, with the exception of the final message, in which the MAC under the symmetric key kw_i derived from a Diffie-Hellman key exchange in the real protocol is replaced by a signature under $\mathsf{sk}(W)$. This change over-approximates the power of the adversary if we assume ephemeral keys are not compromised: to construct a MAC that would be accepted in the last message, the adversary must sign the ephemeral sent in m_2, and so must have compromised $\mathsf{sk}(W)$—but an adversary who compromised $\mathsf{sk}(W)$ and did not generate the ephemeral key would not normally be able to construct the MAC in the real protocol, while they can always construct the signature in our model.

Sound detection. If there is a `detect` event triggered for a particular key pairing, then at least one of the keys in that pairing was compromised before the `detect` event. Formally,

$$\forall k_C, k_W, t_1 \,.\, \mathtt{detect}(\mathsf{pk}(k_C), \mathsf{pk}(k_W)) \;@\; t_1 \Rightarrow \\ (\exists t_0 \,.\, t_0 \lessdot t_1 \wedge \mathtt{compromise}(k_C) \;@\; t_c) \vee (\exists t_0 \,.\, t_0 \lessdot t_1 \wedge \mathtt{compromise}(k_W) \;@\; t_0)\,.$$

Note that this also implies soundness in traces where compromise is restricted to one of the keys.

Agreement without compromise. If no keys were compromised, then a session event by C implies a matching session event by W, with agreement on nonces and counter value. This ensures that the protocol is correct in the uncompromised setting. Since counter values are unique, this also implies injective agreement.

$$\forall keys, data, t_1 \,.\, \mathtt{session}(\text{`C'}, keys, data) \;@\; t_1 \wedge \neg\,(\exists k, t_c \,.\, \mathtt{compromise}(k) \;@\; t_c) \Rightarrow \\ \exists t_0 \,.\, t_0 \lessdot t_1 \wedge \mathtt{session}(\text{`W'}, keys, data) \;@\; t_0.$$

Guaranteed detection of misuse of the CDN key in prior sessions. Assuming the web service's key k_W is not compromised, if W completes a session at time t_3

then either every prior session completed by W has a matching session completed by C, or W will detect misuse of k_C at t_3. Formally,

$$\forall pk_C, pk_W, data_1, data_2, t_0, t_2, t_3 \,.\, t_0 \lessdot t_3 \wedge \texttt{begin}(\text{`C'}, \langle pk_C, pk_W\rangle, data_2) \mathbin{@} t_2 \wedge$$
$$\texttt{session}(\text{`W'}, \langle pk_C, pk_W\rangle, data_2) \mathbin{@} t_3 \wedge \texttt{session}(\text{`W'}, \langle pk_C, pk_W\rangle, data_1) \mathbin{@} t_0 \wedge$$
$$\neg\,(\exists t_c, k_W \,.\, pk_W = \mathsf{pk}(k_W) \wedge \texttt{compromise}(k_W) \mathbin{@} t_c) \Rightarrow$$
$$(\exists t_0 \,.\, \texttt{session}(\text{`C'}, \langle pk_C, pk_W\rangle, data_1) \mathbin{@} t_1) \vee (\texttt{detect}(pk_C, pk_W) \mathbin{@} t_3).$$

Verification of all helper lemmas and trace properties, as well as trace existence lemmas takes approximately two and a half minutes on a four-core Intel i7-6700K processor, using the ‘`I`’ heuristic on our Tamarin branch.

7.1.3 Formal analysis of the Commitment protocol

As with the counter-based protocol above, formal verification of the commitment-based detection protocol shown was performed by constructing a symbolic model in Tamarin. The full model is included in Appendix A.4.1. Analysis of this model relies on the standard symbolic assumptions described in Chapter 2 as well as the same additional PKI assumption used above, so that I and R are aware of each other’s public keys. The model of our commitment-based protocol uses only the `signing` functions and equational theory built into Tamarin, with no additional definitions required.

Agents begin by generating and registering fresh keys through a `RegisterKey` rule, producing `AgentKey` facts. Two such facts are paired through the `BindState_I` and `BindState_R` rules for the I and R roles respectively, which create agent state facts. The state binding rules are restricted to occur only once for each possible role binding for each pair of two registered agent keys, where the two registered keys must not be equal.

The protocol itself is across four rules, with two to express the conditional detection after I receives m_2. These two rules both require that the provided pk_{i-1} verifies the signature on ncc_i, with the additional restriction that the signature on c_{i-1} is, or is not, valid for the normal and detect versions of the rule respectively. The detect version of the rule does not output a valid I state fact, preventing the protocol execution from continuing with that key pairing as would occur with automatic key revocation.

Finally, to augment the adversary’s capability, we include a rule which outputs the

commitment signing key (the only non-public portion of I's state) to the adversary, labelled with a `compromiseState` action. Our final security property is expressed with respect to when this action occurs.

Analyzing this model in TAMARIN required proving a substantial number of reusable lemmas. These include invariants for agent state, linking them back to the point where they were created in the state binding rule, and for the keys used to create commitments. In addition, we have reusable lemmas establishing that commitments are honest unless the key used was compromised, that commitments are unique, and an injectivity property which establishes a strict ordering of rules at the cost of extra case disjunctions. In total, the TAMARIN model requires about four and a half minutes on a four-core Intel i7-6700K processor using the '`I`' heuristic. The analysis proves all helper lemmas, several trace existence properties, and the following security properties.

Sound detection. If there is a `detect` event triggered for a particular key pairing, then the responder's key was compromised before the `detect` event. Formally,

$$\forall k_I, k_R, t_1 \,.\, \texttt{detect}(\mathsf{pk}(k_I), \mathsf{pk}(k_R)) \ @ \ t_1 \Rightarrow (\exists t_0 \,.\, t_0 \lessdot t_1 \wedge \texttt{compromise}(k_R) \ @ \ t_0)\,.$$

Agreement without compromise. If no keys were compromised, then a session event by I implies a matching session event by R, with agreement on the nonce, proof, and commitment. This ensures that the protocol is correct in the uncompromised setting. Since commitments are unique (as proven in one of the helper lemmas), this also implies injective agreement.

$$\forall keys, data, t_1 \,.\, \texttt{session}(\text{`I'}, keys, data) \ @ \ t_1 \wedge \neg\, (\exists k, t_c \,.\, \texttt{compromise}(k) \ @ \ t_c) \Rightarrow \\ \exists t_0 \,.\, t_0 \lessdot t_1 \wedge \texttt{session}(\text{`R'}, keys, data) \ @ \ t_0.$$

Guaranteed detection of misuse of a key in prior sessions. Assuming at most one key is compromised, if I completes a session at t_3, then either every prior session completed by I has a corresponding session completed by R, or I will detect key

misuse. Formally,

$$\forall pk_I, pk_R, data_1, data_2, t_1, t_2, t_3 \;.\; t_1 \lessdot t_2 \lessdot t_3 \wedge \texttt{session}(\text{`R'}, \langle pk_I, pk_R\rangle, data_2) \mathbin{@} t_2 \wedge$$
$$\texttt{session}(\text{`I'}, \langle pk_I, pk_R\rangle, data_2) \mathbin{@} t_3 \wedge \texttt{session}(\text{`I'}, \langle pk_I, pk_R\rangle, data_1) \mathbin{@} t_1 \wedge$$
$$\neg\,(\exists t_c, t_{c2}, k_R, k_I \;.\; pk_R = \mathsf{pk}(k_R) \wedge pk_I = \mathsf{pk}(k_I) \wedge$$
$$\texttt{compromise}(k_R) \mathbin{@} t_c \wedge \texttt{compromise}(k_I) \mathbin{@} t_{c2}) \Rightarrow$$
$$(\exists t_0 \;.\; t_0 \lessdot t_1 \wedge \texttt{session}(\text{`R'}, \langle pk_I, pk_R\rangle, data_1) \mathbin{@} t_0) \vee (\texttt{detect}(pk_I, pk_R) \mathbin{@} t_3).$$

Guaranteed detection of misuse of the responder's key in future sessions. If there was a previous session with agreement and the adversary has not revealed R's state since that session, then a session completed by I will either have a matching session by R or I will detect that there is not. Formally,

$$\forall pk_I, pk_R, data_1, data_2, t_0, t_1, t_3 \;.\; t_0 \lessdot t_1 \lessdot t_3 \wedge$$
$$\texttt{session}(\text{`R'}, \langle pk_I, pk_R\rangle, data_1) \mathbin{@} t_0 \wedge \texttt{session}(\text{`I'}, \langle pk_I, pk_R\rangle, data_1) \mathbin{@} t_1 \wedge$$
$$\texttt{session}(\text{`I'}, \langle pk_I, pk_R\rangle, data_2) \mathbin{@} t_3 \wedge \neg(\texttt{detect}(pk_I, pk_R) \mathbin{@} t_3) \wedge$$
$$\neg\,(\exists t_c, x \;.\; (t_0 \lessdot t_c \lessdot t_3) \wedge \texttt{compromiseState}(\langle pk_I, pk_R\rangle, x)) \Rightarrow$$
$$(\exists t_2 \;.\; t_2 \lessdot t_3 \wedge \texttt{session}(\text{`R'}, \langle pk_I, pk_R\rangle, data_2) \mathbin{@} t_2)\,,$$

7.1.4 Formal analysis of modified 9798-3-3

The model of the modified 9798-3-3 protocol is very similar to that of the commitment-based protocol model in Section 7.1.3, with only minor changes necessary. The 9798-3-3 protocol includes an (unnecessary) nonce generated by the responder and included in the message m_2, and so the session data is updated to reflect that, along with an additional reusable property ordering `session` actions with respect to the time the responder's nonce is generated. All other reusable lemmas are identical, and the adversary is augmented with the same `compromiseState` capability.

The final model takes roughly the same time to analyze as the model of the commitment-based protocol to prove equivalent properties (including helper lemmas and trace existence): about four and a half minutes on a four-core Intel i7-6700K processor, using the '`I`' heuristic. The full model is included in Appendix A.4.2.

Sound detection. If there is a `detect` event triggered for a particular key pairing, then the responder's key was compromised before the `detect` event. Formally,

$$\forall k_I, k_R, t_1 \;.\; \texttt{detect}(\mathsf{pk}(k_I), \mathsf{pk}(k_R)) \;@\; t_1 \Rightarrow (\exists t_0 \;.\; t_0 \lessdot t_1 \wedge \texttt{compromise}(k_R) \;@\; t_0)\,.$$

Agreement without compromise. If no keys were compromised, then a session event by I implies a matching session event by R, with agreement on the nonce, proof, and commitment. This ensures that the protocol is correct in the uncompromised setting. Since commitments are unique (as proven in one of the helper lemmas), this also implies injective agreement.

$$\forall keys, data, t_1 \;.\; \neg\,(\exists k, t_c \;.\; \texttt{compromise}(k) \;@\; t_c) \wedge \texttt{session}(\text{`I'}, keys, data) \;@\; t_1 \Rightarrow \\ \exists t_0 \;.\; t_0 \lessdot t_1 \wedge \texttt{session}(\text{`R'}, keys, data) \;@\; t_0.$$

Guaranteed detection of misuse of a key in prior sessions. Assuming at most one key is compromised, if I completes a session at t_3, then either every prior session completed by I has a corresponding session completed by R, or I will detect key misuse. Formally,

$$\forall pk_I, pk_R, data_1, data_2, t_1, t_2, t_3 \;.\; t_1 \lessdot t_2 \lessdot t_3 \wedge \texttt{session}(\text{`R'}, \langle pk_I, pk_R\rangle, data_2) \;@\; t_2 \wedge \\ \texttt{session}(\text{`I'}, \langle pk_I, pk_R\rangle, data_2) \;@\; t_3 \wedge \texttt{session}(\text{`I'}, \langle pk_I, pk_R\rangle, data_1) \;@\; t_1 \wedge \\ \neg\,(\exists t_c, t_{c2}, k_R, k_I \;.\; pk_R = \mathsf{pk}(k_R) \wedge pk_I = \mathsf{pk}(k_I) \wedge \\ \texttt{compromise}(k_R) \;@\; t_c \wedge \texttt{compromise}(k_I) \;@\; t_{c2}) \Rightarrow \\ (\exists t_0 \;.\; t_0 \lessdot t_1 \wedge \texttt{session}(\text{`R'}, \langle pk_I, pk_R\rangle, data_1) \;@\; t_0) \vee (\texttt{detect}(pk_I, pk_R) \;@\; t_3).$$

Guaranteed detection of misuse of the responder's key in future sessions. If there was a previous session with agreement and the adversary has not revealed R's state since that session, then a session completed by I will either have a matching

session by R or I will detect that there is not. Formally,

$$\begin{aligned}
&\forall pk_I, pk_R, data_1, data_2, t_0, t_1, t_3 \;.\; t_0 \lessdot t_1 \lessdot t_3 \wedge \\
&\quad \texttt{session}(\text{'R'}, \langle pk_I, pk_R\rangle, data_1) \;@\; t_0 \wedge \texttt{session}(\text{'I'}, \langle pk_I, pk_R\rangle, data_1) \;@\; t_1 \wedge \\
&\qquad \texttt{session}(\text{'I'}, \langle pk_I, pk_R\rangle, data_2) \;@\; t_3 \wedge \neg(\texttt{detect}(pk_I, pk_R) \;@\; t_3) \wedge \\
&\qquad\quad \neg\left(\exists t_c, x \;.\; (t_0 \lessdot t_c \lessdot t_3) \wedge \texttt{compromiseState}(\langle pk_I, pk_R\rangle, x)\right) \Rightarrow \\
&\qquad\qquad \left(\exists t_2 \;.\; t_2 \lessdot t_3 \wedge \texttt{session}(\text{'R'}, \langle pk_I, pk_R\rangle, data_2) \;@\; t_2\right),
\end{aligned}$$

7.2 Real-world protocols

In this section we apply our improved modelling techniques and Tamarin modifications in order to perform automated security analysis of two real-world protocols: the key exchange used by WireGuard [35], and the first known security analysis of the full DNP3 Secure Authentication protocol [39].

7.2.1 Analyzing WireGuard

WireGuard is a VPN protocol which aims to provide a secure network tunnel for transporting layer 3 (IP) packets. It has quickly gained popularity, and is poised to take a larger share of VPN traffic in the near future, with Linus Torvalds recently stating “I think we should strive to merge WireGuard and get people moved over to that” [89]. The protocol is extensively detailed in [35], and the key exchange performed is based on the NoiseIK [73] handshake. The WireGuard protocol consists of several mechanisms: an authenticated key exchange, a ‘cookie’ system for mitigating denial of service attacks, and a timer state machine. Notably, WireGuard aims to be silent in the face of unauthenticated packets (“stealthiness”), avoid dynamic memory allocation, avoid parsers, not require any long term storage, and allow initiator and responder to swap roles at any time. These design requirements lead to some subtle adjustments to the NoiseIK key exchange, thus additional formal analysis was an important part of the design.

In this section we will briefly discuss the key exchange and the model before discussing how our improvements to Tamarin show marked increases in analysis speed compared to an original model. Due to the symbolic nature of Tamarin analysis, our work captures the state transitions which may be triggered by the timing

mechanism, but we do not comment on the correctness of the time-based transitions themselves.

WireGuard handshake and model

WireGuard consists of a 1-RTT handshake—from initiator to responder, and responder back to initiator—after which the initiator can then begin an exchange of transport data as shown in Figure 7.1. This initial exchange of transport data from the initiator provides key confirmation, and after the responder has key confirmation either party may send transport data. Transport data is encrypted using an AEAD with a pair of transport keys, one each for sending and receiving, derived from the preceding handshake messages. These are referred to as the *session keys*, for the *session* comprising the handshake and all messages encrypted under those session keys.

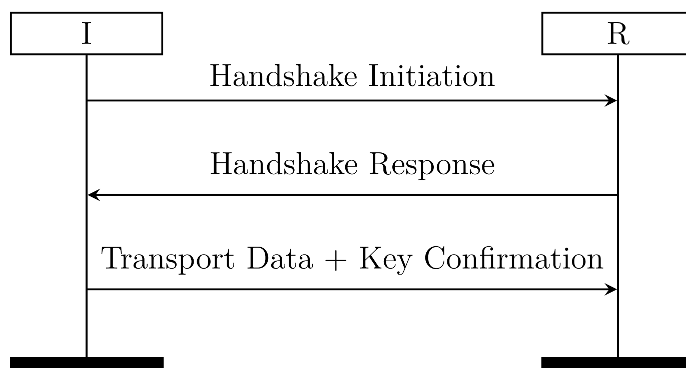

Figure 7.1: 1-RTT handshake between the initiator and responder.

The initiator, i, and the responder, r, both keep several secrets. They each have a static long-term key, S_i and S_r, as well as a random session ephemeral key, E_i and E_r. They may *optionally* also share a pre-shared symmetric key, Q; when Q is in use, WireGuard is using "pre-shared key mode", and otherwise WireGuard is using "normal mode" (where Q is considered to be a fixed string).

The model of WireGuard makes use of the Diffie-Hellman equational theory defined in [64] with five additional formulae: h/1, for hashing, aead/3, decrypt/2, verify/3 for authenticated encryption with additional data (AEAD) operations, and true/0, for an

affirmative result of AEAD decryption verification. These formulae have the equations

$$\text{decrypt}(\text{aead}(k, p, a), k) \to p$$
$$\text{verify}(\text{aead}(k, p, a), a, k) \to \text{true},$$

representing the ability to recover the encrypted plaintext p of $\text{aead}(k, p, a)$ using the key k, and also verify its authenticity using both the key k and the additional authenticated data (AAD) a.

Note that these equations do not provide the adversary a way to scramble the plaintext in the AEAD such that it could be decrypted incorrectly when the AAD is not known, as could happen in a real-world flawed implementation. For the purposes of modelling WireGuard, this is sufficient, since the AAD is always known to both parties. Also note that decrypt is never used by the agents during the handshake as they can both reconstruct the aead term directly; nonetheless, it's necessary for the adversary to learn the plaintext of messages after revealing a key.

In our model we consider an adversary able to compromise three types of secrets at any time, corresponding to the following actions in the trace: $\mathsf{Reveal}(S_i)$ and $\mathsf{Reveal}(S_r)$ correspond to a compromise of either peer's static key, $\mathsf{Reveal}(E_i)$ and $\mathsf{Reveal}(E_r)$ correspond to a compromise of either peer's ephemeral key, and $\mathsf{Reveal}(Q)$ corresponds to a compromise of the preshared key. $\mathsf{Reveal}(Q)$ may also correspond to the preshared key mode not being used at all, in which case the WireGuard protocol uses an all zero key, which is equivalent to the adversary simply knowing it in advance.

WireGuard assumes that agents share their public keys and optionally a symmetric key before the key exchange occurs. Thus, rather than receiving public keys over the adversary-controlled network, agents add the public key of their peers directly by taking in $!\mathsf{AgentKey}(k)$ and $!\mathsf{AgentPSK}(k)$ facts directly in the premises. These facts are persistent, as multiple agents might have the same peer, and are created by rules which generate fresh values and—in the case of agent keys—reveal the corresponding public key directly to the adversary. This is equivalent to peers receiving public and preshared keys out-of-band before the protocol begins. Note that agents are identified only by their public keys in WireGuard, and attacks on any sort of PKI used to receive these keys or associate these keys to particular identities are out of scope.

In the model, agent state contains a few terms invariant over all rules: the identifier the agent associates with the current state, the pre-shared key, their long term key, the public key of the other agent, and finally the corresponding static-static key computed

from these two. The latter is not necessary to store in state—it could be computed from their long term key and the public key—but ‘caching’ it in this way reduces the number of Diffie-Hellman variants that need to be precomputed by TAMARIN.

Prior to our analysis, WireGuard had two slightly different handshakes depending on whether there was a PSK set or not. This made formal analysis more complex, with two different modes and a method for switching between them. Through discussion with Jason Donenfeld about our formal analysis efforts, it was decided that simplifying the modes would allow for better formal analysis, simpler code, and potentially confer privacy benefits (through, e.g. observational equivalence). This discussion was also taken into account by the Noise developers, and upstreamed into the Noise specification in [73, Revision 32b].

The full model and all properties can be found in Appendix A.1.

Benchmarking WireGuard analysis

WireGuard is an ideal protocol for benchmarking some of the performance improvements we obtain through our modelling techniques and TAMARIN modifications. Unlike the DNP3 example in Section 7.2.2, the WireGuard protocol has only the single set of invariant terms; thus, the model remains tractable for TAMARIN to analyze even without our modifications, using a basic inductive lemma. Figure 7.2 shows the effects of the various methods of expressing the invariant property, both with and without the modified precomputation described in Section 6.2.2 (note that without an invariant property the analysis does not terminate). We also compare our `'I'` heuristic described in Section 6.1 against the base `'S'` heuristic. Note that all of these models make use of heuristic annotations as described in Section 6.1.2 to mark the looping initial rules as low priority.

The timings shown in Figure 7.2 are the average of 10 consecutive runs on a server with 32 virtual cores (comprising two hyperthreaded 8-core Sandy Bridge Xeon processors at 2.60GHz), including both precomputation and analysis, with standard error. The lemma used for the invariant property is simply

$$\forall inv, t_1 \,.\, \texttt{Invariants}(inv) \;@\; t_1 \Rightarrow \exists t_0 \,.\, (t_0 < t_1) \wedge \texttt{InvariantSource}(inv) \;@\; t_0,$$

marked as either `sources`, `reuse` or both in our benchmarks. The invariant premise formulation is written using the modelling technique described in Section 6.2.2: a

persistent fact !StateInvariants() is output when agent state is created, and included in the premises of all later rules which make use of terms in the invariant.

As expected, explicitly adding the invariant in the rule premises performs similarly to a `sources` lemma with action precomputation, since these two both incorporate the invariant in all precomputed cases. The modified heuristic also performs consistently better across all cases.

7.2.2 Analyzing DNP3 Secure Authentication

Most of the world's power grids are monitored and controlled remotely. In practice, power grids are controlled by transmitting monitoring and control messages between authorized operators ('users') that send commands from control centres, and substations or remote devices ('outstations'). The messages may be passed over a range of different media, such as direct serial connections, Ethernet, Wi-Fi, or un-encrypted radio links. As a consequence, we cannot assume that these channels guarantee confidentiality or authenticity.

The commands that are passed over these media are critical to the security of the power grid: they can make changes to operating parameters such as increases or decreases in voltage, opening or closing valves, or starting or stopping motors [33]. It is therefore desirable that an adversary in control of one of these media links should not be able to insert or modify messages. This has motivated the need for a way to authenticate received messages.

The DNP3 standard, more formally known as IEEE 1815-2012, the "Standard for Electric Power Systems Communications – Distributed Network Protocol" [39], is used by most of the world's power grids for communication, and increasingly for other utilities such as water and gas.

Secure Authentication version 5 (SAv5) is a new protocol family within DNP3, and was standardized in 2012 (Chapter 7 of IEEE 1815-2012 [39], based on IEC/TS 62351-5 [74]). SAv5's goal is to provide authenticated communication between parties within a utility grid. Given the security-critical nature of the power grid, one might expect that DNP3: SAv5 would have attracted substantial scrutiny. Instead, there had been very little analysis before our recent work using Tamarin [26]. The reason for this is readily apparent; the DNP3: SAv5 protocol is inherently complex, comprising three interacting sub-protocols that each maintain state and update the keys used in other sub-protocols. The resulting state machines have multiple interacting looping

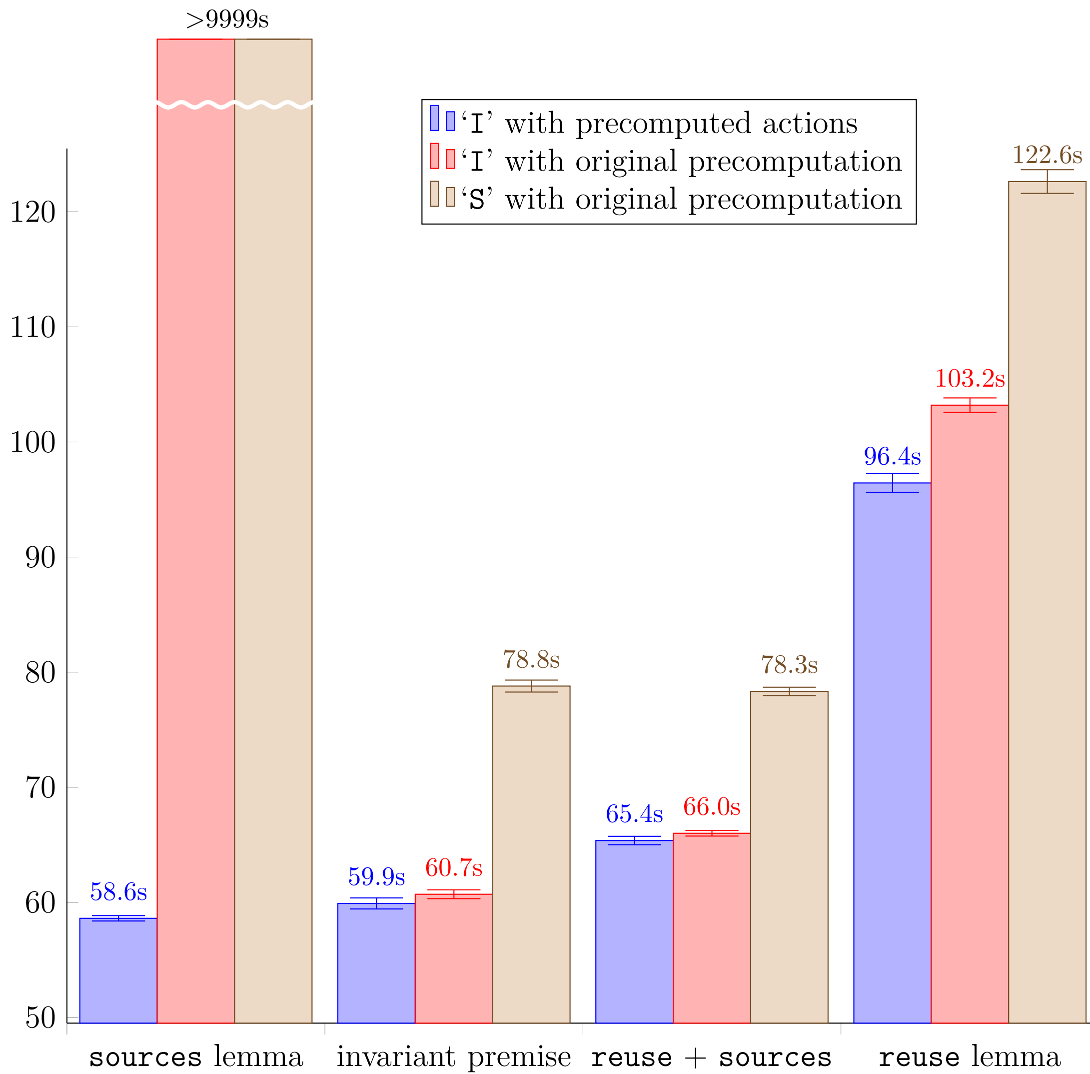

Figure 7.2: Four variants of the WireGuard model benchmarked showing the improvement from our 'I' heuristic described in Section 6.1, and from adding precomputed action goals as discussed in Section 6.2.2. Values shown are the mean of 10 consecutive runs, with standard error shown.

constructions which pose a substantial challenge for protocol security analysis tools. It has long been known in theory [49] that analyzing each of the interacting sub-protocols in isolation would not be sufficient to prove security—and indeed, there are practical attacks that exploit cross-protocol interactions in other protocols [11, 62]. Thus, it was necessary to model all interacting protocols to have confidence in our security analysis.

The DNP3 standard [39] gives both high level and semi-formal descriptions to serve as an implementation guide, as well as providing an informal problem statement and conformance guidelines. The Secure Authentication v5 protocol itself is described in Chapter 7 of [39]. Here, we briefly give an overview of the interacting sub-protocols to give intuition for the difficulty involved, before explaining how the techniques above were applied to perform analysis of the full protocol.

The SAv5 sub-protocols

There are three types of actor in SAv5: the *Authority*, the *Users* (operating from a Master station), and the *Outstations*. The Authority decides who are legitimate users, and generates new (medium-term) Update Keys for these users. Users send control packets to outstations, who act upon them if they are successfully authenticated. Outstations send back (similarly authenticated) monitoring packets. Each user can communicate with multiple outstations, and each outstation can communicate with multiple users. Users regularly generate new (short-term) *Session Keys* for each direction of this communication, and transport these keys to the outstations. Session keys are distributed and updated using long-term *Authority Keys* and medium-term *Update keys*. These three different keys are used by three sub-protocols: the Session Key Update protocol, the Critical ASDU Authentication protocol, and the Update Key Change protocol. Figure 7.3 shows a conceptual summary of the relationship between the sub-protocols.

Initial Key Distribution Before any protocols are run, a long-term Authority Key and an initial medium-term update key must be pre-distributed to each party. These keys are distributed "over a secure channel" (e. g. via USB stick) to the respective parties. The Session Keys are *not* pre-distributed.

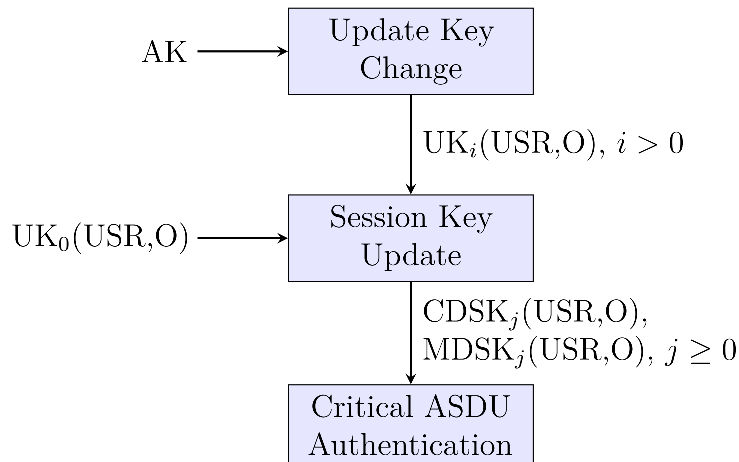

Figure 7.3: The conceptual relationships between the keys used in the sub-protocols of DNP3:SAv5, with flow of keys between them (vertical) and required pre-shared keys (horizontal).

Session Key Update Before parties can exchange control or monitoring messages, the user and outstation must initialize session keys. This sub-protocol initializes (and later updates) a new, symmetric Session Key for each communication direction.

After approximately 15 minutes or 1,000 critical messages (both configurable) the session keys will expire. The user and outstation run the Session Key Update Protocol again, where the user generates fresh symmetric session keys, and sends them to the outstation, encrypted with their current update key. These session keys *must* remain secret, but the secrecy of new keys importantly does not rely on the secrecy of previous session keys, only on the secrecy of the current update key.

Critical ASDU Authentication Outstations use this sub-protocol to verify that received control packets were genuinely sent by a legitimate user. Vice-versa, this sub-protocol allows a user to confirm that received monitoring packets were genuinely sent by a legitimate outstation. As this is an authentication-only protocol, critical ASDUs are not confidential. After this sub-protocol's first execution, the faster 'Aggressive Mode' may be performed: this cuts the non-aggressive mode's three messages to just one by sending the ASDU and a keyed HMAC in the same message.

Update Key Change After a longer time, the update key may expire. The user and outstation (helped by the Authority) will execute the Update Key Change Protocol. A new update key is created by the Authority, and sent to both the user and outstation.

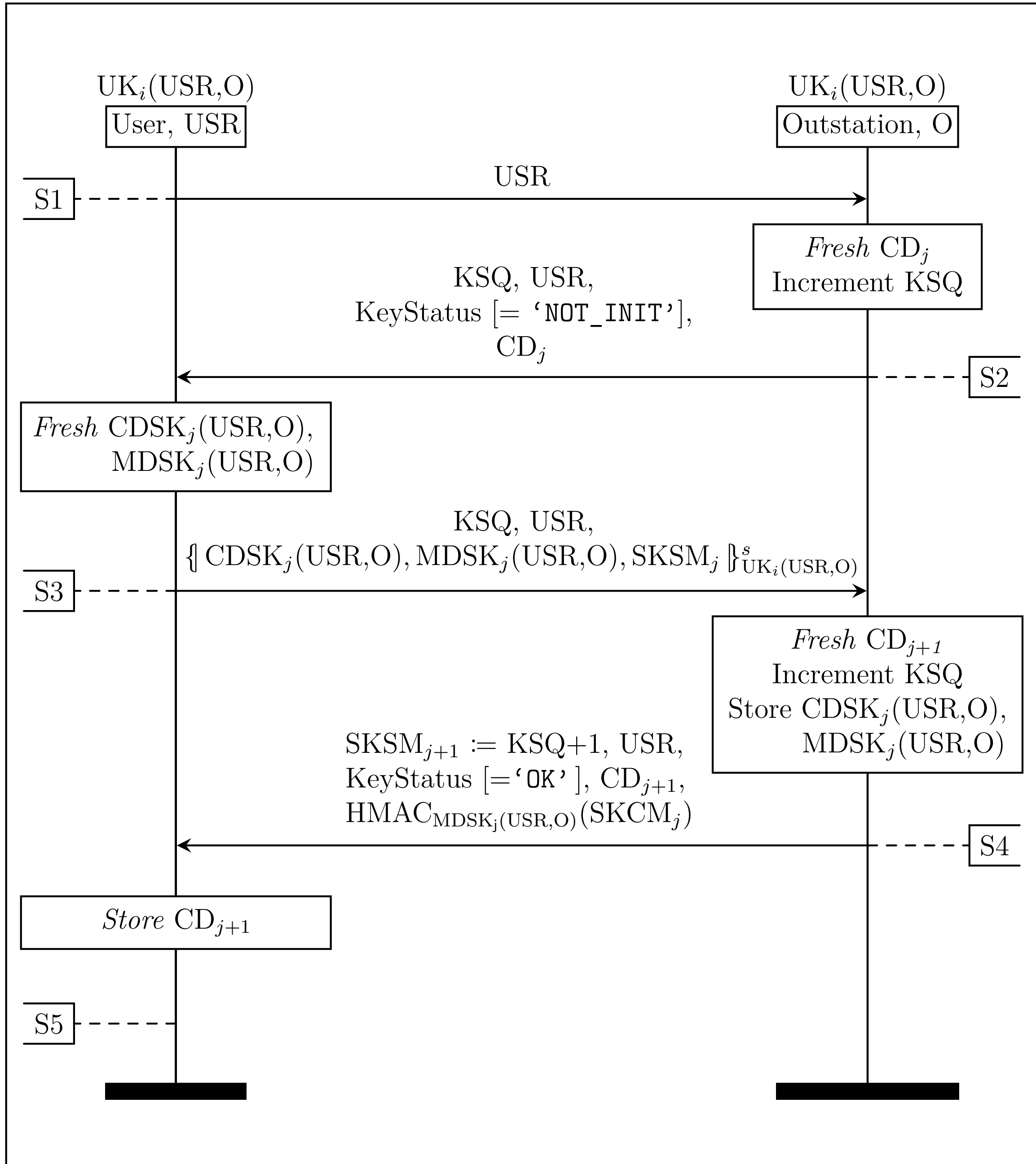

Figure 7.4: The full Session Key Update Protocol, an example of one of the DNP3:SAv5 subprotocols. The labels S1–5 identify the state transitions that occur during the protocol. This protocol can also begin at rule S3, using the previously provided key sequence number (KSQ) and challenge data (CD) values stored at S5, if the key status has not been changed since.

Complexity for automated analysis

Much of the complexity within Secure Authentication v5 comes from the transitions between different stateful sub-protocols as well as the multiple directions of the ASDU Authentication Protocol. Each of these sub-protocols updates some part of agent state, and each of them rely on parts of state updated by other sub-protocols.

For example, the ability for an outstation to receive an aggressive mode request depends on:

1. their pairing to a user, set at initialization and invariant across all subprotocols;
2. their current state machine state, which must be set to `SecurityIdle` or `WaitForReply`. This status is invariant through the Update Key Change Protocol but modified by the other two subprotocols;
3. their current session keys, set by the previously completed Session Key Update Protocol, and invariant across the other two subprotocols;
4. the status of those session keys, which is invariant across all subprotocols but can be changed at any time through a key expiry state transition;
5. the 'current' challenge from the user, modified when a non-aggressive mode request was successfully verified by the outstation, which is invariant in the aggressive mode protocol variant and all other subprotocols;
6. and the number of aggressive mode requests received by the outstation since the current challenge was set, which is invariant in all other subprotocols.

Each of these parts of state can be updated independently of each other, and each is invariant over several subprotocols.

The state machine described in the protocol specification [39] captures very little of the protocol logic; the current state machine state is only one of the six dependencies listed above, and in fact is invariant for the outstation through both the Update Key Change Protocol and Session Key Update Protocol subprotocols. An accurate Tamarin model must include this much larger range of transitions, as well as their associated errors and timeouts; it is not possible to directly map the states and transitions in the specification to those in the model. Consequently, the difficulty of interpreting the specification is greatly increased.

Tamarin model

All three sub-protocols' rules and interactions were modelled as rules in Tamarin's operational semantics, and can be found in Appendix A.2. The final model comprises 30 multiset rewriting rules in approximately 450 SLoC, with 10 types of invariants represented as actions within these rules, while others are captured in the state transitions themselves.

Invariants within the model include:

1. the authority key and relevant identifiers for both the user and outstation as assigned during the initial key distribution,
2. update key invariants for both the user and outstation,
3. session key invariants for both the user and outstation,
4. and the last challenge sent or received in each direction for both the user and the outstation.

Additionally, there is a 'last key status message' which is stored by the outstation in both the $S2$ and $S4$ rules. Although this message is invariant through all other rules, it is only used in rules where it is also modified, so we can efficiently represent it with a linear fact consumed and output by those two rules. Finally, there are three 'keys to reveal' facts output whenever new keys are generated, which are used to model adversary compromise and represented with persistent facts. The combined invariant relations between protocol rules in the model included in Appendix A.2 are shown in Figure 7.5, which shows edges between rules in which invariants are set and rules in which those invariants are used. In other words, each edge depicts a relation between two rules executed by the same agent where the head relies on at least one term in the tail, but the agent may execute an unbounded number of other rules between them.

In our model, we make use of all improvements described throughout Chapter 6. Heuristic annotations are used extensively—for example, facts representing current user and outstation state are annotated to prioritize those with few possible originating rules. All ten action invariants are proven using `sources` lemmas, with redundant goals reduced through the technique described in Section 6.2.3.

The model verifies automatically in approximately 6.5 minutes on a server with 32 virtual cores comprising two hyperthreaded 8-core Sandy Bridge Xeon processors at 2.60GHz, including both precomputation and analysis. Unlike the WireGuard case study above, it was not possible to perform automatic verification of our DNP3

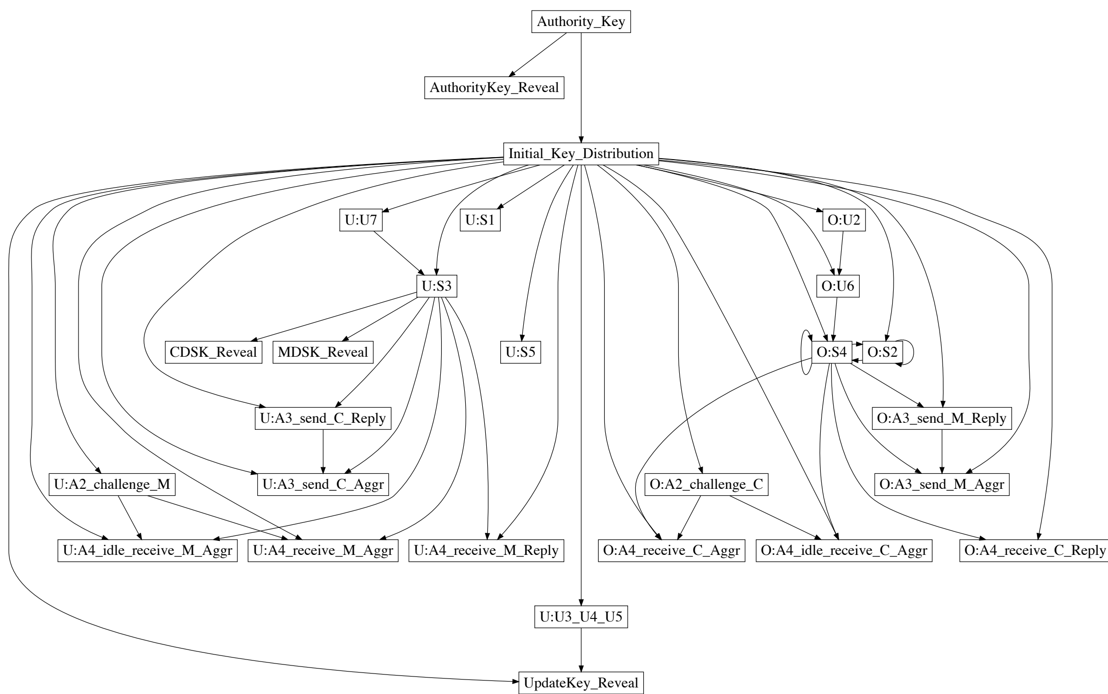

Figure 7.5: Protocol rules and loop invariants in the DNP3 model. Rules executed by the user and outstation are prefixed by 'U:' and 'O:' respectively, and invariants are shown with edges from the rules that set their value to the rules that use them. For example, the U:A3_send_C_Aggressive rule uses the $CDSK$ invariant from the U:S3 rule, the challenge data invariant since the last U:A3_send_C_Reply rule, and the identifiers set up in the Initial_Key_Distribution rule.

model without our modifications to precomputation unless we introduced explicit invariant constraints as persistent premises, as was done for the model referenced in our paper [26], to avoid the need for a modified Tamarin to reproduce our results.

With invariant constraints in rule premises, the model can be automatically verified in approximately two minutes, as this model implicitly assumes invariants that must be proven in the original. Persistent facts are also handled more efficiently by Tamarin and do not require the additional work to handle redundant goals described in Section 6.2.3, indicating that there is room for further improvements to Tamarin's handling of action goals.

Chapter 8

Conclusion

We have shown that soundly detecting the misuse of secrets is achievable, even without complex systems. In the future, we foresee detection-related security properties being a valuable tool for protocol designers to enhance the resilience of their protocols against key compromise. We believe that this thesis provides the intuition necessary to consider detection-related security properties, and that it strengthens our ability to formally verify them.

In practical scenarios, our modifications to existing protocols show that it is possible to provide sound detection with little overhead. Our results also imply that detection of secret misuse can be added to many existing systems with minor protocol updates, rather than requiring additional, independent infrastructure like a transparency overlay. The case studies we analyzed—Keyless SSL and smartcard authentication—are only two examples out of a great many domains in which authenticating agents have the opportunity to verify the consistency or causality of key uses.

From a theoretical perspective, developing the foundations of—and intuition for—detection is valuable both for future protocol design, and in reexamining existing designs with new insights. The intuition we have developed revealed a missed opportunity in the most common designs of transparency overlays. Although easy to miss during the engineering of a complex system, the potential for improvement is clear when considering the design within a broader theory of detection.

Our work successfully improved Tamarin's ability to analyze stateful protocols and aided in the automated analysis of our protocol designs and modification to real-world protocols. It also enabled us to perform the first formal analysis of the complex DNP3 Secure Authentication protocol, and WireGuard, an increasingly popular tunnelling

protocol. Going forward, our improvements will facilitate formal verification of protocols by allowing even more to be modelled and analyzed in TAMARIN.

8.1 Future directions

Our work presents several possibilities for future study, both in the applications of misuse detection without false positives and in further improving TAMARIN's ability to analyze stateful protocols. Here, we present two directions that we consider particularly compelling.

8.1.1 Auditable wiretapping

A number of nations have adopted policy which broadly legalizes the use of weaponized signals intelligence techniques on their citizens where deemed appropriate. In some cases, such as recent Swiss law, active malicious interference is subject to federal court approval and restricted to a fixed timeline before additional court action is required (see [54] for a recent overview of national laws).

In most cases, the process of lawful interception is entirely opaque to citizens. Even when legislation requires fixed timelines for surveillance, it is usually not possible to verify that these timelines have been followed; an investigation can make use of parallel construction to conceal illegally gathered evidence. And indeed, a report by Reuters in 2013 [82] found that the U.S. Drug Enforcement Agency directly advised agents to perform parallel construction to conceal warrantless surveillance.

Misuse detection allows for the detection of adversarial activity once there is no longer active interception of communication. Combined with a form of post-compromise secrecy [24], detecting a divergence of session keys would allow agents to determine that there was a loss of confidentiality. With adequate information, agents could pinpoint the session in which their keys diverged.

This opens the possibility of designing systems in which wiretapping is auditable *a posteriori* by the agents whose communications were intercepted. Such a system would ensure that the eavesdropper must actively intercept all messages received and sent by the agent. Surveillance would not be immediately detected, thus still allowing for lawful interception, but once it ceases (or the case is brought to court) the parties targeted could compare the appropriate state and verify that the surveillance was performed legally.

8.1.2 Further improvements to Tamarin

The analysis of stateful protocols in Tamarin is substantially easier than it was when the tool was first developed, but there is still room for improvement. In our work analyzing DNP3's Secure Authentication, loop invariants were manually discovered and transcribed to rule actions, with many of these actions repeated verbatim in nearly every rule. Within the field of program verification there has been substantial work on automatic inference of loop invariants, to address the tedius and complex process of manually identifying invariants. Tamarin would greatly benefit by applying this work; precomputation to automatically infer and annotate invariants would be a valuable addition.

In addition, there are several common constructions in modern protocols which are based around nested functions. For example, protocols aiming for post-compromise security generally derive each new key from a combination of the previous key and some new key material, yielding keys roughly of the form $kdf(k_i \| kdf(k_{i-1} \| \ldots))$. In Tamarin currently, it is difficult to prove any properties inductively over such constructions; there is no way to express a property which holds over arbitrary depths of nesting. Concretely, although we can prove properties like

$$K(k_i) \Rightarrow K(k_{i-1})$$

for two consecutive keys k_{i-1} and k_i, it would not be possible to apply this inductively to prove that

$$K(k_i) \Rightarrow K(k_0)$$

for some earlier initial key k_0. Working with constructions like hash chains or nested key derivation requires careful modelling to get around these constraints wherever possible. Some initial progress has been made in this direction, with an approach to handling iterated applications of a function to the same term in [83], but further work is needed to allow properties over nested functions with many terms.

Bibliography

[1] 3GGP2. *N.S0005-0: Cellular Radio Telecommunications Intersystem Operations.* 1991.

[2] Martín Abadi and Roger M. Needham. "Prudent Engineering Practice for Cryptographic Protocols". In: *IEEE Trans. Software Eng.* 22.1 (1996), pp. 6–15. URL: `https://doi.org/10.1109/32.481513`.

[3] Secretary of the Air Force. *AFI 91-104: Nuclear Surety Tamper Control and Detection Programs.* Air Force Instruction. United States Air Force, 2013.

[4] David J Alvarez. *Allied and Axis signals intelligence in World War II.* Cass series–studies in intelligence. London: Cass, 1999.

[5] Myrto Arapinis, Joshua Phillips, Eike Ritter, and Mark Dermot Ryan. "StatVerif: Verification of stateful processes". In: *Journal of Computer Security* 22.5 (2014), pp. 743–821. URL: `https://doi.org/10.3233/JCS-140501`.

[6] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. "The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications". In: *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings.* Vol. 3576. Lecture Notes in Computer Science. Springer, 2005, pp. 281–285. URL: `https://doi.org/10.1007/11513988_27`.

[7] David A. Basin and Cas J. F. Cremers. "Modeling and Analyzing Security in the Presence of Compromising Adversaries". In: *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings.* Vol. 6345. Lecture Notes

in Computer Science. Springer, 2010, pp. 340–356. URL: https://doi.org/10.1007/978-3-642-15497-3_21.

[8] David A. Basin, Cas J. F. Cremers, and Simon Meier. "Provably Repairing the ISO/IEC 9798 Standard for Entity Authentication". In: *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings.* Vol. 7215. Lecture Notes in Computer Science. Springer, 2012, pp. 129–148. URL: https://doi.org/10.1007/978-3-642-28641-4_8.

[9] David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. "ARPKI: Attack Resilient Public-Key Infrastructure". In: *Proc. ACM Conference on Computer and Communications Security.* Scottsdale, Arizona, USA, 2014, pp. 382–393. URL: http://doi.acm.org/10.1145/2660267.2660298.

[10] Gary Belvin. *Key Transparency Overview.* https://github.com/google/keytransparency/blob/master/docs/overview.md. 2016.

[11] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. "Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS". In: *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014.* IEEE Computer Society, 2014, pp. 98–113. URL: https://doi.org/10.1109/SP.2014.14.

[12] Bruno Blanchet. "Security protocols: from linear to classical logic by abstract interpretation". In: *Inf. Process. Lett.* 95.5 (2005), pp. 473–479. URL: https://doi.org/10.1016/j.ipl.2005.05.011.

[13] Erik-Oliver Blass, Kaoutar Elkhiyaoui, and Refik Molva. "Tracker: Security and Privacy for RFID-based Supply Chains". In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011.* The Internet Society, 2011. URL: http://www.isoc.org/isoc/conferences/ndss/11/pdf/9_1.pdf.

[14] Richard R. Brooks, P. Y. Govindaraju, Matthew Pirretti, Narayanan Vijaykrishnan, and Mahmut T. Kandemir. "On the Detection of Clones in Sensor Networks Using Random Key Predistribution". In: *IEEE Trans. Systems, Man,*

and Cybernetics, Part C 37.6 (2007), pp. 1246–1258. URL: https://doi.org/10.1109/TSMCC.2007.905824.

[15] Richard S. Canter. *Lock with security counter.* US Patent 3,789,639. Feb. 1974. URL: https://patentscope.wipo.int/search/en/detail.jsf?docId=US36708044.

[16] Michael R Carr. "Smart card technology with case studies". In: *Proc. International Carnahan Conference on Security Technology.* IEEE. 2002, pp. 158–159.

[17] Iliano Cervesato. "Typed Multiset Rewriting Specifications of Security Protocols". In: *Electr. Notes Theor. Comput. Sci.* 40 (2000), pp. 8–51. URL: https://doi.org/10.1016/S1571-0661(05)80035-0.

[18] Iliano Cervesato, Nancy A. Durgin, Patrick Lincoln, John C. Mitchell, and Andre Scedrov. "A Meta-Notation for Protocol Analysis". In: *Proceedings of the 12th IEEE Computer Security Foundations Workshop, CSFW 1999, Mordano, Italy, June 28-30, 1999.* IEEE Computer Society, 1999, pp. 55–69. URL: https://doi.org/10.1109/CSFW.1999.779762.

[19] Melissa Chase and Sarah Meiklejohn. "Transparency Overlays and Applications". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016.* ACM, 2016, pp. 168–179. URL: http://doi.acm.org/10.1145/2976749.2978404.

[20] Laurent Chuat, Pawel Szalachowski, Adrian Perrig, Ben Laurie, and Eran Messeri. "Efficient gossip protocols for verifying the consistency of Certificate logs". In: *2015 IEEE Conference on Communications and Network Security, CNS 2015, Florence, Italy, September 28-30, 2015.* IEEE, 2015, pp. 415–423. URL: https://doi.org/10.1109/CNS.2015.7346853.

[21] Jeremy Clark and Paul C. van Oorschot. "SoK: SSL and HTTPS: Revisiting Past Challenges and Evaluating Certificate Trust Model Enhancements". In: *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013.* IEEE Computer Society, 2013, pp. 511–525. URL: https://doi.org/10.1109/SP.2013.41.

[22] CloudFlare. *Keyless SSL: The Nitty Gritty Technical Details.* https://blog.cloudflare.com/keyless-ssl-the-nitty-gritty-technical-details/. 2014.

[23] Katriel Cohn-Gordon, Cas J. F. Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. "A Formal Security Analysis of the Signal Messaging Protocol". In: *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*. IEEE, 2017, pp. 451–466. URL: https://doi.org/10.1109/EuroSP.2017.27.

[24] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. "On Post-Compromise Security". In: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 2016, pp. 164–178. URL: https://doi.org/10.1109/CSF.2016.19.

[25] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. "On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees". In: *IACR Cryptology ePrint Archive* 2017 (2017), p. 666. URL: http://eprint.iacr.org/2017/666.

[26] Cas Cremers, Martin Dehnel-Wild, and Kevin Milner. "Secure Authentication in the Grid: A Formal Analysis of DNP3: SAv5". In: *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings*. Lecture Notes in Computer Science. To appear. Springer, 2017.

[27] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. "A Comprehensive Symbolic Analysis of TLS 1.3". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. ACM, 2017, pp. 1773–1788. URL: http://doi.acm.org/10.1145/3133956.3134063.

[28] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication". In: *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 2016, pp. 470–485. URL: https://doi.org/10.1109/SP.2016.35.

[29] United States Department of Defense. *National Industrial Security Program operating manual*. United States (DoD, DoE, NRC, CIA), 2016.

[30] Stéphanie Delaune, Steve Kremer, Mark Dermot Ryan, and Graham Steel. "A Formal Analysis of Authentication in the TPM". In: *Formal Aspects of Security and Trust - 7th International Workshop, FAST 2010, Pisa, Italy, September*

16-17, 2010. Revised Selected Papers. Vol. 6561. Lecture Notes in Computer Science. Springer, 2010, pp. 111–125. URL: https://doi.org/10.1007/978-3-642-19751-2_8.

[31] Grit Denker and Jonathan K. Millen. "Modeling Group Communication Protocols Using Multiset Term Rewriting". In: *Electr. Notes Theor. Comput. Sci.* 71 (2002), pp. 20–39. URL: https://doi.org/10.1016/S1571-0661(05)82527-7.

[32] John Dilley, Bruce M. Maggs, Jay Parikh, Harald Prokop, Ramesh K. Sitaraman, and William E. Weihl. "Globally Distributed Content Delivery". In: *IEEE Internet Computing* 6.5 (2002), pp. 50–58. URL: https://doi.org/10.1109/MIC.2002.1036038.

[33] DNP Users Group. *A DNP3 Protocol Primer (Revision A)*. 2005. URL: https://www.dnp.org/AboutUs/DNP3%5C%20Primer%5C%20Rev%5C%20A.pdf.

[34] Danny Dolev and Andrew C. Yao. *On the security of public key protocols*. Tech. rep. Stanford University, May 1981. URL: http://dl.acm.org/citation.cfm?id=891726.

[35] Jason A. Donenfeld. "WireGuard: Next Generation Kernel Network Tunnel". In: *24nd Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2016*. The Internet Society, 2017. URL: https://www.internetsociety.org/sites/default/files/ndss2017_04A-3_Donenfeld_paper.pdf.

[36] Jason Donenfeld and Kevin Milner. *Formal Verification of the WireGuard Protocol*. 2017. URL: https://www.wireguard.com/papers/wireguard-formal-verification.pdf.

[37] Jannik Dreier, Charles Duménil, Steve Kremer, and Ralf Sasse. "Beyond Subterm-Convergent Equational Theories in Automated Verification of Stateful Protocols". In: *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Vol. 10204. Lecture Notes in Computer Science. Springer, 2017, pp. 117–140. URL: https://doi.org/10.1007/978-3-662-54455-6_6.

[38] Jannik Dreier, Maxime Puys, Marie-Laure Potet, Pascal Lafourcade, and Jean-Louis Roch. “Formally Verifying Flow Properties in Industrial Systems”. In: *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications (ICETE 2017) - Volume 4: SECRYPT, Madrid, Spain, July 24-26, 2017.* SciTePress, 2017, pp. 55–66. URL: https://doi.org/10.5220/0006396500550066.

[39] *Electric Power Systems Communications – Distributed Network Protocol (DNP3).* Standard 1815-2012. IEEE Standards Association, Oct. 2012.

[40] *Entity authentication – Part 3: Mechanisms using digital signature techniques.* Standard ISO/IEC 9798-3:1998. International Organization for Standardization, 1998.

[41] Santiago Escobar, Catherine A. Meadows, and José Meseguer. “A rewriting-based inference system for the NRL Protocol Analyzer and its meta-logical properties”. In: *Theor. Comput. Sci.* 367.1-2 (2006), pp. 162–202. URL: https://doi.org/10.1016/j.tcs.2006.08.035.

[42] Santiago Escobar, Ralf Sasse, and José Meseguer. “Folding variant narrowing and optimal variant termination”. In: *J. Log. Algebr. Program.* 81.7-8 (2012), pp. 898–928. URL: https://doi.org/10.1016/j.jlap.2012.01.002.

[43] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert van Renesse. “Bitcoin-NG: A Scalable Blockchain Protocol”. In: *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016.* USENIX Association, 2016, pp. 45–59. URL: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eyal.

[44] Matthew K. Franklin. “A survey of key evolving cryptosystems”. In: *IJSN* 1.1/2 (2006), pp. 46–53. URL: https://doi.org/10.1504/IJSN.2006.010822.

[45] Gregor Gössler and Daniel Le Métayer. “A general trace-based framework of logical causality”. In: *International Workshop on Formal Aspects of Component Software.* Springer. 2013, pp. 157–173.

[46] Johan Håstad, Jakob Jonsson, Ari Juels, and Moti Yung. “Funkspiel schemes: an alternative to conventional tamper resistance”. In: *CCS 2000, Proceedings of the 7th ACM Conference on Computer and Communications Security, Athens,*

Greece, November 1-4, 2000. ACM, 2000, pp. 125–133. URL: http://doi.acm.org/10.1145/352600.352619.

[47] Clemens Helfmeier, Dmitry Nedospasov, Christopher Tarnovsky, Jan Starbug Krissler, Christian Boit, and Jean-Pierre Seifert. "Breaking and entering through the silicon". In: *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. ACM, 2013, pp. 733–744. URL: http://doi.acm.org/10.1145/2508859.2516717.

[48] Gene Itkis. "Cryptographic tamper evidence". In: *Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS 2003, Washington, DC, USA, October 27-30, 2003*. ACM, 2003, pp. 355–364. URL: http://doi.acm.org/10.1145/948109.948156.

[49] John Kelsey, Bruce Schneier, and David A. Wagner. "Protocol Interactions and the Chosen Protocol Attack". In: *Security Protocols, 5th International Workshop, Paris, France, April 7-9, 1997, Proceedings*. Vol. 1361. Lecture Notes in Computer Science. Springer, 1997, pp. 91–104. URL: https://doi.org/10.1007/BFb0028162.

[50] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil D. Gligor. "Accountable key infrastructure (AKI): a proposal for a public-key validation infrastructure". In: *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013*. International World Wide Web Conferences Steering Committee / ACM, 2013, pp. 679–690. URL: http://dl.acm.org/citation.cfm?id=2488448.

[51] Paul C. Kocher. "On Certificate Revocation and Validation". In: *Financial Cryptography, Second International Conference, FC'98, Anguilla, British West Indies, February 23-25, 1998, Proceedings*. Vol. 1465. Lecture Notes in Computer Science. Springer, 1998, pp. 172–177. URL: https://doi.org/10.1007/BFb0055481.

[52] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. "Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing". In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. USENIX Association, 2016, pp. 279–296. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kogias.

[53] Gerhard de Koning Gans, Jaap-Henk Hoepman, and Flavio D. Garcia. “A Practical Attack on the MIFARE Classic”. In: *Smart Card Research and Advanced Applications, 8th IFIP WG 8.8/11.2 International Conference, CARDIS 2008, London, UK, September 8-11, 2008. Proceedings*. Vol. 5189. Lecture Notes in Computer Science. Springer, 2008, pp. 267–282. URL: https://doi.org/10.1007/978-3-540-85893-5_20.

[54] Douwe Korff, Ben Wagner, Julia Powles, Renata Avila, and Ulf Buermeyer. *Boundaries of Law: Exploring Transparency, Accountability, and Oversight of Government Surveillance Regimes*. Legal Studies Research Paper Series Paper No. 16/2017. University of Cambridge Faculty of Law, 2017.

[55] Steve Kremer and Robert Künnemann. “Automated analysis of security protocols with global state”. In: *Journal of Computer Security* 24.5 (2016), pp. 583–616. URL: https://doi.org/10.3233/JCS-160556.

[56] Steve Kremer, Mark Ryan, and Ben Smyth. “Election Verifiability in Electronic Voting Protocols”. In: *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings*. Vol. 6345. Lecture Notes in Computer Science. Springer, 2010, pp. 389–404. URL: https://doi.org/10.1007/978-3-642-15497-3_24.

[57] Robert Künnemann. “Automated Backward Analysis of PKCS#11 v2.20”. In: *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*. Vol. 9036. Lecture Notes in Computer Science. Springer, 2015, pp. 219–238. URL: https://doi.org/10.1007/978-3-662-46666-7_12.

[58] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. “Accountability: definition and relationship to verifiability”. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*. ACM, 2010, pp. 526–535. URL: http://doi.acm.org/10.1145/1866307.1866366.

[59] Ben Laurie, Adam Langley, and Emilia Kasper. *Certificate Transparency*. RFC 6962. RFC Editor, June 2013. URL: http://www.rfc-editor.org/rfc/rfc6962.

[60] Mikko Lehtonen, Florian Michahelles, and Elgar Fleisch. “How to detect cloned tags in a reliable way from incomplete RFID traces”. In: *Proc. IEEE Conference on RFID*. 2009, pp. 257–264.

[61] Junrong Liu, Yu Yu, François-Xavier Standaert, Zheng Guo, Dawu Gu, Wei Sun, Yijie Ge, and Xinjun Xie. “Small Tweaks Do Not Help: Differential Power Analysis of MILENAGE Implementations in 3G/4G USIM Cards”. In: *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I*. Vol. 9326. Lecture Notes in Computer Science. Springer, 2015, pp. 468–480. URL: `https://doi.org/10.1007/978-3-319-24174-6_24`.

[62] Nikos Mavrogiannopoulos, Frederik Vercauteren, Vesselin Velichkov, and Bart Preneel. “A cross-protocol attack on the TLS protocol”. In: *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. ACM, 2012, pp. 62–72. URL: `http://doi.acm.org/10.1145/2382196.2382206`.

[63] Simon Meier. *Advancing automated security protocol verification*. 2012. URL: `http://dx.doi.org/10.3929/ethz-a-009790675`.

[64] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. “The TAMARIN Prover for the Symbolic Analysis of Security Protocols”. In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 696–701. URL: `https://doi.org/10.1007/978-3-642-39799-8_48`.

[65] Kevin Milner. *Protocol models contained in this thesis*. 2018. URL: `http://thesis.kamilner.ca`.

[66] Kevin Milner. *Tamarin fork containing precomputation changes and annotations*. 2018. URL: `https://github.com/kmilner/tamarin-prover/tree/actionprecomp-and-annotes`.

[67] Kevin Milner, Cas Cremers, Jiangshan Yu, and Mark Ryan. “Automatically Detecting the Misuse of Secrets: Foundations, Design Principles, and Applications”. In: *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, 2017, pp. 203–216. URL: `https://doi.org/10.1109/CSF.2017.21`.

[68] Moxie Marlinspike and Trevor Perrin. *The Double Ratchet Algorithm*. https://whispersystems.org/docs/specifications/doubleratchet/. 2016.

[69] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2009.

[70] Moni Naor and Kobbi Nissim. "Certificate revocation and certificate update". In: *IEEE Journal on Selected Areas in Communications* 18.4 (2000), pp. 561–570. URL: https://doi.org/10.1109/49.839932.

[71] Roger M. Needham and Michael D. Schroeder. "Using Encryption for Authentication in Large Networks of Computers". In: *Commun. ACM* 21.12 (1978), pp. 993–999. URL: http://doi.acm.org/10.1145/359657.359659.

[72] David Oswald and Christof Paar. "Breaking Mifare DESFire MF3ICD40: Power Analysis and Templates in the Real World". In: *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*. Vol. 6917. Lecture Notes in Computer Science. Springer, 2011, pp. 207–222. URL: https://doi.org/10.1007/978-3-642-23951-9_14.

[73] Trevor Perrin. *The Noise Protocol Framework*. 2016. URL: http://noiseprotocol.org/noise.pdf.

[74] *Power systems management and associated information exchange – Data and communications security – Part 5: Security for IEC 60870-5 and derivatives*. Standard IEC/TS 62351-5:2013. International Electrotechnical Commission, 2013.

[75] Josyula R. Rao, Pankaj Rohatgi, Helmut Scherzer, and Stephane Tinguely. "Partitioning Attacks: Or How to Rapidly Clone Some GSM Cards". In: *2002 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 12-15, 2002*. IEEE Computer Society, 2002, pp. 31–41. URL: https://doi.org/10.1109/SECPRI.2002.1004360.

[76] Masoud Rostami, Farinaz Koushanfar, and Ramesh Karri. "A Primer on Hardware Security: Models, Methods, and Metrics". In: *Proceedings of the IEEE* 102.8 (2014), pp. 1283–1295. URL: https://doi.org/10.1109/JPROC.2014.2335155.

[77] Mark Dermot Ryan. “Enhanced Certificate Transparency and End-to-End Encrypted Mail”. In: *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. URL: http://www.internetsociety.org/doc/enhanced-certificate-transparency-and-end-end-encrypted-mail.

[78] Mark Ryan, Myrto Arapinis, and Eike Ritter. *StatVerif: Verification of stateful processes*. 2017. URL: http://markryan.eu/research/statverif/.

[79] Kazue Sako and Joe Kilian. “Receipt-Free Mix-Type Voting Scheme - A Practical Solution to the Implementation of a Voting Booth”. In: *Advances in Cryptology - EUROCRYPT ’95, International Conference on the Theory and Application of Cryptographic Techniques, Saint-Malo, France, May 21-25, 1995, Proceeding*. Vol. 921. Lecture Notes in Computer Science. Springer, 1995, pp. 393–403. URL: https://doi.org/10.1007/3-540-49264-X_32.

[80] Benedikt Schmidt. *Formal analysis of key exchange protocols and physical protocols*. 2012. URL: http://dx.doi.org/10.3929/ethz-a-009898924.

[81] Benedikt Schmidt, Simon Meier, Cas J. F. Cremers, and David A. Basin. “Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties”. In: *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. IEEE Computer Society, 2012, pp. 78–94. URL: https://doi.org/10.1109/CSF.2012.25.

[82] John Shiffman and Kristina Cooke. *Exclusive: U.S. directs agents to cover up program used to investigate Americans*. https://www.reuters.com/article/us-dea-sod/exclusive-u-s-directs-agents-to-cover-up-program-used-to-investigate-americans-idUSBRE97409R20130805. Aug. 2013.

[83] Cedric Staub. “Adding support for user-defined sorts and sorted function symbols to Tamarin”. MSc Thesis. ETH Zürich, 2013. URL: https://www.research-collection.ethz.ch/handle/20.500.11850/73721.

[84] Pawel Szalachowski, Stephanos Matsumoto, and Adrian Perrig. “PoliCert: Secure and Flexible TLS Certificate Management”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. ACM, 2014, pp. 406–417. URL: http://doi.acm.org/10.1145/2660267.2660355.

[85] Tamarin Prover Team. *Main source code repository of the Tamarin prover for security protocol verification, Rev. 8e82369.* URL: https://github.com/tamarin-prover/tamarin-prover/tree/8e823691ad3325bce8921617b013735523d74557.

[86] The Tamarin Team. *Tamarin Prover.* 2018. URL: http://tamarin-prover.github.io.

[87] The Tamarin Team. *Tamarin Prover Manual. Security Protocol Analysis in the Symbolic Model.* May 19, 2017. URL: https://tamarin-prover.github.io/manual/ (visited on 05/19/2017).

[88] *The Zettabyte Era: Trends and Analysis.* White paper. Cisco, June 2017.

[89] Linus Torvalds. *Re: [PATCH 00/31 v2] PTI support for x86_32.* Feb. 2018. URL: https://lkml.org/lkml/2018/2/13/752.

[90] Trusted Computing Group. *TPM 1.2 Specification.* https://www.trustedcomputinggroup.org/tpm-1-2-protection-profile/. July 2014.

[91] United States Department of Defense. *DoD Common Access Card.* http://www.cac.mil/common-access-card/. May 2016.

[92] Roel Verdult, Flavio D. Garcia, and Josep Balasch. "Gone in 360 Seconds: Hijacking with Hitag2". In: *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012.* USENIX Association, 2012, pp. 237–252. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/verdult.

[93] Jorden Whitefield, Liqun Chen, Frank Kargl, Andrew Paverd, Steve Schneider, Helen Treharne, and Stephan Wesemeyer. "Formal Analysis of V2X Revocation Protocols". In: *Security and Trust Management - 13th International Workshop, STM 2017, Oslo, Norway, September 14-15, 2017, Proceedings.* Vol. 10547. Lecture Notes in Computer Science. Springer, 2017, pp. 147–163. URL: https://doi.org/10.1007/978-3-319-68063-7_10.

[94] John Wilkes. *Detector Lock with Key (c. 1675 – c. 1700).* Rijksmuseum, Object BK-NM-886. URL: http://hdl.handle.net/10934/RM0001.COLLECT.14828.

[95] Christopher Wingert and Mullaguru Naidu. *CDMA 1XRTT security overview.* Tech. rep. Qualcomm Incorporated, 2002.

[96] Peter Wright and Paul Greengrass. *Spycatcher: The Candid Autobiography of a Senior Intelligence Officer*. Dell, 1987.

[97] Christos Xenakis and Lazaros F. Merakos. “Security in Third Generation Mobile Networks”. In: *Computer Communications* 27.7 (2004), pp. 638–650. URL: `https://doi.org/10.1016/j.comcom.2003.12.004`.

[98] Jiangshan Yu, Vincent Cheval, and Mark Ryan. “DTKI: a new formalized PKI with no trusted parties”. In: *CoRR* abs/1408.1023 (2014). URL: `http://arxiv.org/abs/1408.1023`.

[99] Jiangshan Yu, Mark Ryan, and Cas Cremers. “DECIM: Detecting Endpoint Compromise In Messaging”. In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 486.

[100] David Zanetti, Leo Fellmann, and Srdjan Capkun. “Privacy-preserving clone detection for RFID-enabled supply chains”. In: *Proc. IEEE Conference on RFID*. 2010, pp. 37–44.

[101] Davide Zanetti, Srdjan Capkun, and Ari Juels. “Tailing RFID Tags for Clone Detection”. In: *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013. URL: `http://internetsociety.org/doc/tailing-rfid-tags-clone-detection`.

Appendix A

Full symbolic models

The improvements described in Chapter 6 are not upstreamed into the main development branch at the time of writing. A fork of TAMARIN containing an implementation of the improvements described in Chapter 6 can be found at [66].

For convenience, a digital copy of all files can be found at [65].

A.1 WireGuard

A.1.1 Model with invariants in premises

```
/*
 * Protocol:     Wireguard protocol
 * Modeler:      Kevin Milner & Jason Donenfeld
 * Date:         2017
 * Source:       Original
 */

theory WireGuard
begin

builtins: hashing, diffie-hellman
/* Normally an aead would be arity 4, but in the handshake the nonce is always
 * the fixed value 0 so for legibility we do not include it */
functions: aead/3, decrypt/2, verify/3, true/0
/* The plaintext can be recovered with the key */
equations: decrypt(aead(k, p, a), k) = p
/* The authentication can be checked with the key and AAD */
equations: verify(aead(k, p, a), a, k) = true

/* This restriction tells Tamarin that whenever an Eq( ) fact occurs,
 * the terms in it must be equal. This allows us to model rules that
 * only trigger if e.g. the AEAD correctly verifies */
restriction Eq_testing: "All x y #i. Eq(x, y) @ i ==> x = y"

/* === Setup and key reveal rules === */

/* Keygen is separate from pairing to allow keys to be paired
 * more than once (if they were generated fresh in the pairing
 * this would not be possible */
rule AgentKeyGen:
    [ Fr(~ltk) ]
    --[DHKey(~ltk)]->
    [!F_AgentKey(~ltk), Out('g'^~ltk)]

/* Semi-malicious or ignorant agents might share a PSK with multiple
 * parties, so we overapproximate this by allowing the same PSK to be reused
 * arbitrarily. If this finds an attack then it may not be a 'real'
 * attack, but if it doesn't then there is no problem with PSK reuse. */
rule PSKKeyGen:
    [ Fr(~psk) ]
    --[PSKey(~psk)]->
    [ !F_AgentPSK(~psk) ]

/* Key Reveals for our adversary model. These allow the adversary to reveal
 * any of the keys below at any time, unless restricted in the lemma we wish
 * to prove. */
rule PSK_reveal:
   [ !F_AgentPSK(k) ] --[ Reveal_PSK(k) ]-> [ Out(k) ]

rule AgentKey_reveal:
   [ !F_AgentKey(k) ] --[ Reveal_AK('g'^k) ]-> [ Out(k) ]

rule EphKey_reveal:
   [ !EphKeytoReveal(k) ] --[ Reveal_EphK('g'^k) ]-> [ Out(k) ]

/* Models an agent adding anothers public key out-of-band, we assume that
 * all relationships set up this way are 'sane' and both of the keys involved
 * were generated fresh. */
rule AddPublicKey:
    let pkB = 'g'^~ltkB in
    [ !F_AgentKey(~ltkA)
    , !F_AgentKey(~ltkB)
    , !F_AgentPSK(~psk)
    , Fr(~id)
    ]-->
    [ /* For search efficiency, state is divided into
       * an invariant portion and a variant portion. This
       * allows tamarin to immediately bind the keys back
       * to this initial pairing rule. The fresh ~id is used
       * to identify this pairing. We cache the SiSr in the
       * invariant to reduce Tamarin's variant precomputation time. */
      State(~id,'no_message_state')
    , !F_StateInvariants(~id,~psk,~ltkA,pkB,pkB^~ltkA)
    ]

/* === Message Rules === */

rule Handshake_Init:
    let pkI    = 'g'^~ltkI
```

```
80           pekI   = 'g'^~ekI
81           eisr   = pkR^~ekI
82           /* Constructing the init message m1: */
83           cii    = h('noise')
84           hii    = h(<cii, 'id', pkR, pekI>)
85           ci0    = h(<cii, pekI, '1'>)
86           ci1    = h(<ci0, eisr, '1'>)
87           ki1    = h(<ci0, eisr, '2'>)
88           astat  = aead(ki1, <pkI, ~pkISurrogate>, hii)
89           hi0    = h(<hii, astat>)
90           ci2    = h(<ci1, sisr, '1'>)
91           ki2    = h(<ci1, sisr, '2'>)
92           ats    = aead(ki2, $ts, hi0)
93           hi1    = h(<hi0, ats>)
94           /* NOTE: MACs used in the DDoS protection are not modeled, so we assume
95            * they are some arbitrary public values (i.e. known to the adversary,
96            * like a fixed string). */
97           m1     = <'1', ~sidI, pekI, astat, ats, $mac1, $mac2> in
98       [ /* Init can be triggered at any time, even when currently performing
99          * another role or on another stage. This could happen e.g. because of
100         * a timeout. As such we bind the previous state as 'anything' and
101         * discard it. */
102        State(~id,anything)[-]
103      , !F_StateInvariants(~id,~psk,~ltkI,pkR,sisr)
104      , Fr(~sidI)
105      , Fr(~ekI)
106      /* The pkISurrogate is used later to approximate identity hiding. */
107      , Fr(~pkISurrogate)
108      ]-->
109      [ State(~id,<'init',~sidI,~ekI,ci2,ki2,hi1>)
110      , !EphKeytoReveal(~ekI)
111      , Out(m1)
112      ]
113
114  rule Handshake_Resp:
115      let pkR    = 'g'^~ltkR
116          pekR   = 'g'^~ekR
117          eisr   = pekI^~ltkR
118          eier   = pekI^~ekR
119          sier   = pkI^~ekR
120          /* Reconstructing what should be in m1: */
121          cri    = h('noise')
122          hri    = h(<cri, 'id', pkR, pekI>)
123          cr0    = h(<cri, pekI, '1'>)
124          cr1    = h(<cr0, eisr, '1'>)
125          kr1    = h(<cr0, eisr, '2'>)
126          astat  = aead(kr1, <pkI, pkISurrogate>, hri)
127          hr0    = h(<hri, astat>)
128          cr2    = h(<cr1, sisr, '1'>)
129          kr2    = h(<cr1, sisr, '2'>)
130          ats    = aead(kr2, $ts, hr0)
131          hr1    = h(<hr0, ats>)
132          m1     = <'1', sidI, pekI, astat, ats, $mac1, $mac2>
133          /* Constructing the response message m2: */
134          cr3    = h(<cr2, pekR, '1'>)
135          hr2    = h(<hr1, pekR>)
136          cr4    = h(<cr3, eier, '1'>)
137          cr5    = h(<cr4, sier, '1'>)
138          cr6    = h(<cr5, ~psk, '1'>)
139          hrt    = h(<cr5, ~psk, '2'>)
140          kr6    = h(<cr5, ~psk, '3'>)
141          hr3    = h(<hr2, hrt>)
142          aempt  = aead(kr6, 'e', hr3)
143          hr4    = h(<hr3, aempt>)
144          m2     = <'2', sidI, ~sidR, pekR, aempt, $mac1, $mac2> in
145      [ State(~id,anything)[-]
146      , !F_StateInvariants(~id,~psk,~ltkR,pkI,sisr)
147      , Fr(~ekR)
148      , Fr(~sidR)
149      , In(m1)
150      ]--[
151          /* In the real protocol, this isn't actually cr6, but is rather
152           * derived from it via some more hashing, but it doesn't make a
153           * difference here -- if the adversary knows cr6,
154           * it can compute the derived key. */
155          RKeys(<pkI, pkR, pekI, pekR, ~psk, cr6>)
156        , Identity_Surrogate(pkISurrogate)
157      ]->
158      [ State(~id,<'transport_unconfirmed',~sidR,pekI,pekR,cr6>)
159      , !EphKeytoReveal(~ekR)
160      , Out(m2)
161      ]
162
163  rule Handshake_Complete:
164      let pkI    = 'g'^~ltkI
```

```
165          pekI   = 'g'^~ekI
166          eier   = pekR^~ekI
167          sier   = pekR^~ltkI
168          /* Reconstruct what should be in m2: */
169          ci3    = h(<ci2, pekR, '1'>)
170          hi2    = h(<hi1, pekR>)
171          ci4    = h(<ci3, eier, '1'>)
172          ci5    = h(<ci4, sier, '1'>)
173          ci6    = h(<ci5, ~psk, '1'>)
174          hit    = h(<ci5, ~psk, '2'>)
175          ki6    = h(<ci5, ~psk, '3'>)
176          hi3    = h(<hi2, hit>)
177          aempt = aead(ki6, 'e', hi3)
178          hi4    = h(<hi3, aempt>)
179          m2     = <'2', sidI, sidR, pekR, aempt, $mac1, $mac2>
180          /* We assume the first transport message is 'as weak as possible',
181           * that is, all values in it are public other than the key used to
182           * encrypt. */
183          mtr    = <'4', sidR, aead(ci6, 'message', '0')> in
184      [ State(~id,<'init',sidI,~ekI,ci2,ki2,hi1>)[+]
185      , !F_StateInvariants(~id,~psk,~ltkI,pkR,sisr)
186      , In(m2)
187      ]--[
188        /* In the real protocol, this isn't actually ci6, but is rather derived
189         * from it via some more hashing, but it doesn't make a difference here
190         * (as with cr6 above). */
191        IKeys(<pkI, pkR, pekI, pekR, ~psk, ci6>)
192      ]->
193      [ State(~id,<'transport',sidI,ci6>)
194      , Out(mtr)
195      ]
196
197  /* === Key Confirmation ===
198   *
199   * For proving agreement properties after the first transport message
200   * This is not faithfully modelled, since we don't care about the contents
201   * of the transport message--instead it's generically 'anything encrypted
202   * with the transport key I just set up'. */
203
204  rule R_ConfirmedTransportMessage:
205      let mtr = <'4', sid, data> in
206      [ State(~id,<'transport_unconfirmed',sidR,pekI,pekR,cr>)[+]
207      , !F_StateInvariants(~id,~psk,~ltkR,pkI,sisr)
208      , In(mtr)
209      ]--[
210        RConfirm(<pkI, 'g'^~ltkR, pekI, pekR, ~psk, cr>)
211        //We manually verify this instead of the recomputation used for other
212        //terms above, since morally R might not know the contents of the message.
213      , Eq(verify(data, '0', cr), true)
214      ]->
215      [ State(~id,<'transport',sidR,cr>) ]
216
217  /* === Session Traces ===
218   *
219   * We show that at least the protocol works. */
220
221  lemma exists_session: exists-trace
222      "Ex pki pkr peki pekr psk ck #i #j.
223          IKeys(<pki, pkr, peki, pekr, psk, ck>) @ i
224          & RConfirm(<pki, pkr, peki, pekr, psk, ck>) @ j & #i < #j"
225
226  /* There are only three numbers, 0, 1, and infinity. So 2 == infinity */
227  lemma exists_two_sessions: exists-trace
228      "Ex pki pkr sesskeys sesskeys2 #i #j #i2 #j2.
229          IKeys(<pki, pkr, sesskeys>) @ i
230          & RConfirm(<pki, pkr, sesskeys>) @ j & #i < #j
231          & IKeys(<pki, pkr, sesskeys2>) @ i2
232          & RConfirm(<pki, pkr, sesskeys2>) @ j2 & #i2 < #j2
233          & #i < #i2 & not(sesskeys = sesskeys2)"
234
235
236  /* === Agreement Properties === */
237
238  lemma I_disagreement_implies_Sr_or_SiEi_compromise_and_PSK_compromise[reuse]:
239      "All pki pkr peki pekr psk ck #i.
240          /* If I believes they have completed a handshake with R */
241          IKeys(<pki, pkr, peki, pekr, psk, ck>) @ i
242          /* but R doesn't have a matching session */
243          & not(Ex #j. #j < #i & RKeys(<pki, pkr, peki, pekr, psk, ck>) @ j)
244      ==> /* Then the PSK was compromised (or not in use) and */
245          (Ex #j. Reveal_PSK(psk) @ j & #j < #i) & (
246          /* either R's static was compromised */
247          (Ex #j. Reveal_AK(pkr) @ j & #j < #i)
248          /* or both I's static and ephemeral were already compromised */
249          | (Ex #j #j2. Reveal_AK(pki) @ j & #j < #i
250              & Reveal_EphK(peki) @ j2 & #j2 < #i)
```

```
251         )"
252
253 lemma R_disagreement_implies_Si_or_SrEr_compromise_and_PSK_compromise[reuse]:
254     "All pki pkr peki pekr psk ck #i.
255         /* If R believes they have a confirmed session with I */
256         RConfirm(<pki, pkr, peki, pekr, psk, ck>) @ i
257         /* but I doesn't have a matching session */
258         & not(Ex #j. #j < #i & IKeys(<pki, pkr, peki, pekr, psk, ck>) @ j)
259     ==> /* Then the PSK was compromised (or not in use) and */
260         (Ex #j. Reveal_PSK(psk) @ j & #j < #i) & (
261         /* either I's static was compromised */
262         (Ex #j. Reveal_AK(pki) @ j & #j < #i)
263         /* or both R's static and ephemeral were already compromised */
264         | (Ex #j #j2. Reveal_AK(pkr) @ j & Reveal_EphK(pekr) @ j2
265             & #j2 < #i & #j < #i)
266         )"
267
268 lemma UKS_resistance:
269     "All pki1 pki2 pkr1 pkr2 peki1 peki2 pekr1 pekr2 psk1 psk2 ck #i #j.
270         IKeys(<pki1, pkr1, peki1, pekr1, psk1, ck>) @ i
271         & RKeys(<pki2, pkr2, peki2, pekr2, psk2, ck>) @ j
272     ==> pki1 = pki2 & pkr1 = pkr2 & peki1 = peki2 & pekr1 = pekr2 & psk1 = psk2"
273
274 lemma session_uniqueness:
275     /* For both I and R, a 'confirmed session key' will always be unique
276      * (even if all keys are compromised). This follows from I and R generating
277      * random ephemeral values. Note that this could be violated if the
278      * RNG can be controlled instead of just revealed, which we do not model. */
279     "(All pki pkr peki pekr psk ck #i.
280         IKeys(<pki, pkr, peki, pekr, psk, ck>) @ i
281     ==> not(Ex peki2 pekr2 #k.
282             IKeys(<pki, pkr, peki2, pekr2, psk, ck>) @ k & not(#k = #i)))
283     &(All pki pkr peki pekr psk ck #i.
284         RConfirm(<pki, pkr, peki, pekr, psk, ck>) @ i
285     ==> not(Ex peki2 pekr2 psk2 #k.
286             RConfirm(<pki, pkr, peki2, pekr2, psk2, ck>) @ k & not(#k = #i)))"
287
288 /* === Secrecy Properties === */
289
290 lemma key_secrecy[reuse]:
291     "(All pki pkr peki pekr psk ck #i #i2.
292         /* If I and R agree on keys */
293         IKeys(<pki, pkr, peki, pekr, psk, ck>) @ i
294         & RKeys(<pki, pkr, peki, pekr, psk, ck>) @ i2
295     ==> /* Either the adversary doesn't know the keys */
296         not(Ex #j. K(ck) @ j)
297         /* or the psk was compromised (or not in use) */
298         | ((Ex #j. Reveal_PSK(psk) @ j) & (
299            /* and pair of keys from the same agent was revealed (at any time) */
300            (Ex #j #j2. Reveal_AK(pki) @ j & Reveal_EphK(peki) @ j2)
301          | (Ex #j #j2. Reveal_AK(pkr) @ j & Reveal_EphK(pekr) @ j2))
302         ))"
303
304 lemma identity_hiding:
305     /* Since the public static is always symbolically known to the adversary,
306      * we represent identity hiding by whether the adversary could learn some
307      * other value that is put in the first message at the
308      * same position as the public static. */
309     "All pki pkr peki pekr ck surrogate #i #j.
310         /* If R received a handshake init with a particular identity surrogate */
311         RKeys(<pki, pkr, peki, pekr, ck>) @ i & Identity_Surrogate(surrogate) @ i
312         /* And the adversary knows that surrogate for the identity */
313         & K(surrogate) @ j
314     ==> /* Then adversary compromised at least one of
315          * R's static, I's static, or I's ephemeral */
316         (Ex #j. Reveal_AK(pkr) @ j)
317         | (Ex #j. Reveal_AK(pki) @ j)
318         | (Ex #j. Reveal_EphK(peki) @ j)"
319
320 end
```

A.1.2 Model with sources lemmas

```
1 /*
2  * Protocol:    Wireguard protocol
3  * Modeler:     Kevin Milner & Jason Donenfeld
4  * Date:        2017
5  * Source:      Original
6  */
7
8 theory WireGuard
```

```
9   begin
10
11  builtins: hashing, diffie-hellman
12  /* Normally an aead would be arity 4, but in the handshake the nonce is always
13   * the fixed value 0 so for legibility we do not include it */
14  functions: aead/3, decrypt/2, verify/3, true/0
15  /* The plaintext can be recovered with the key */
16  equations: decrypt(aead(k, p, a), k) = p
17  /* The authentication can be checked with the key and AAD */
18  equations: verify(aead(k, p, a), a, k) = true
19
20  /* This restriction tells Tamarin that whenever an Eq( ) fact occurs,
21   * the terms in it must be equal. This allows us to model rules that
22   * only trigger if e.g. the AEAD correctly verifies */
23  restriction Eq_testing: "All x y #i. Eq(x, y) @ i ==> x = y"
24
25  /* === Setup and key reveal rules === */
26
27  /* Keygen is separate from pairing to allow keys to be paired
28   * more than once (if they were generated fresh in the pairing
29   * this would not be possible */
30  rule AgentKeyGen:
31      [ Fr(~ltk) ]
32      --[DHKey(~ltk)]->
33      [!F_AgentKey(~ltk), Out('g'^~ltk)]
34
35  /* Semi-malicious or ignorant agents might share a PSK with multiple
36   * parties, so we overapproximate this by allowing the same PSK to be reused
37   * arbitrarily. If this finds an attack then it may not be a 'real'
38   * attack, but if it doesn't then there is no problem with PSK reuse. */
39  rule PSKKeyGen:
40      [ Fr(~psk) ]
41      --[PSKey(~psk)]->
42      [ !F_AgentPSK(~psk) ]
43
44  /* Key Reveals for our adversary model. These allow the adversary to reveal
45   * any of the keys below at any time, unless restricted in the lemma we wish
46   * to prove. */
47  rule PSK_reveal:
48     [ !F_AgentPSK(k) ] --[ Reveal_PSK(k) ]-> [ Out(k) ]
49
50  rule AgentKey_reveal:
51     [ !F_AgentKey(k) ] --[ Reveal_AK('g'^k) ]-> [ Out(k) ]
52
53  rule EphKey_reveal:
54     [ !EphKeytoReveal(k) ] --[ Reveal_EphK('g'^k) ]-> [ Out(k) ]
55
56  /* Models an agent adding anothers public key out-of-band, we assume that
57   * all relationships set up this way are 'sane' and both of the keys involved
58   * were generated fresh. */
59  rule AddPublicKey:
60      let pkB    = 'g'^~ltkB
61          invar  = <~psk,~ltkA,pkB,pkB^~ltkA>
62      in
63      [ !F_AgentKey(~ltkA)
64      , !F_AgentKey(~ltkB)
65      , !F_AgentPSK(~psk)
66      , Fr(~id)
67      ]--[
68        InvariantSource(~id,invar)
69      ]->
70      [ /* For search efficiency, state is divided into
71         * an invariant portion and a variant portion. This
72         * allows tamarin to immediately bind the keys back
73         * to this initial pairing rule. The fresh ~id is used
74         * to identify this pairing. We cache the SiSr in the
75         * invariant to reduce Tamarin's variant precomputation time. */
76        State(~id,invar,'no_message_state')
77      ]
78
79  /* === Message Rules === */
80
81  rule Handshake_Init:
82      let invar = <~psk,~ltkI,pkR,sisr>
83          pkI    = 'g'^~ltkI
84          pekI   = 'g'^~ekI
85          eisr   = pkR^~ekI
86          /* Constructing the init message m1: */
87          cii    = h('noise')
88          hii    = h(<cii, 'id', pkR, pekI>)
89          ci0    = h(<cii, pekI, '1'>)
90          ci1    = h(<ci0, eisr, '1'>)
91          ki1    = h(<ci0, eisr, '2'>)
92          astat = aead(ki1, <pkI, ~pkISurrogate>, hii)
93          hi0    = h(<hii, astat>)
94          ci2    = h(<ci1, sisr, '1'>)
95          ki2    = h(<ci1, sisr, '2'>)
```

```
96          ats    = aead(ki2, $ts, hi0)
97          hi1    = h(<hi0, ats>)
98          /* NOTE: MACs used in the DDoS protection are not modeled, so we assume
99           * they are some arbitrary public values (i.e. known to the adversary,
100          * like a fixed string). */
101         m1     = <'1', ~sidI, pekI, astat, ats, $mac1, $mac2> in
102      [ /* Init can be triggered at any time, even when currently performing
103         * another role or on another stage. This could happen e.g. because of
104         * a timeout. As such we bind the previous state as 'anything' and
105         * discard it. */
106        State(~id,invar,anything)[-]
107      , Fr(~sidI)
108      , Fr(~ekI)
109      /* The pkISurrogate is used later to approximate identity hiding. */
110      , Fr(~pkISurrogate)
111      ]--[
112        Invariant(~id,invar)
113      ]->
114      [ State(~id,invar,<'init',~sidI,~ekI,ci2,ki2,hi1>)
115      , !EphKeytoReveal(~ekI)
116      , Out(m1)
117      ]
118
119 rule Handshake_Resp:
120     let invar = <~psk,~ltkR,pkI,sisr>
121         pkR    = 'g'^~ltkR
122         pekR   = 'g'^~ekR
123         eisr   = pekI^~ltkR
124         eier   = pekI^~ekR
125         sier   = pkI^~ekR
126         /* Reconstructing what should be in m1: */
127         cri    = h('noise')
128         hri    = h(<cri, 'id', pkR, pekI>)
129         cr0    = h(<cri, pekI, '1'>)
130         cr1    = h(<cr0, eisr, '1'>)
131         kr1    = h(<cr0, eisr, '2'>)
132         astat  = aead(kr1, <pkI, pkISurrogate>, hri)
133         hr0    = h(<hri, astat>)
134         cr2    = h(<cr1, sisr, '1'>)
135         kr2    = h(<cr1, sisr, '2'>)
136         ats    = aead(kr2, $ts, hr0)
137         hr1    = h(<hr0, ats>)
138         m1     = <'1', sidI, pekI, astat, ats, $mac1, $mac2>
139         /* Constructing the response message m2: */
140         cr3    = h(<cr2, pekR, '1'>)
141         hr2    = h(<hr1, pekR>)
142         cr4    = h(<cr3, eier, '1'>)
143         cr5    = h(<cr4, sier, '1'>)
144         cr6    = h(<cr5, ~psk, '1'>)
145         hrt    = h(<cr5, ~psk, '2'>)
146         kr6    = h(<cr5, ~psk, '3'>)
147         hr3    = h(<hr2, hrt>)
148         aempt  = aead(kr6, 'e', hr3)
149         hr4    = h(<hr3, aempt>)
150         m2     = <'2', sidI, ~sidR, pekR, aempt, $mac1, $mac2> in
151      [ State(~id,invar,anything)[-]
152      , Fr(~ekR)
153      , Fr(~sidR)
154      , In(m1)
155      ]--[
156         /* In the real protocol, this isn't actually cr6, but is rather
157          * derived from it via some more hashing, but it doesn't make a
158          * difference here -- if the adversary knows cr6,
159          * it can compute the derived key. */
160         RKeys(<pkI, pkR, pekI, pekR, ~psk, cr6>)
161       , Identity_Surrogate(pkISurrogate)
162       , Invariant(~id,invar)
163      ]->
164      [ State(~id,invar,<'transport_unconfirmed',~sidR,pekI,pekR,cr6>)
165      , !EphKeytoReveal(~ekR)
166      , Out(m2)
167      ]
168
169 rule Handshake_Complete:
170     let invar = <~psk,~ltkI,pkR,sisr>
171         pkI    = 'g'^~ltkI
172         pekI   = 'g'^~ekI
173         eier   = pekR^~ekI
174         sier   = pekR^~ltkI
175         /* Reconstruct what should be in m2: */
176         ci3    = h(<ci2, pekR, '1'>)
177         hi2    = h(<hi1, pekR>)
178         ci4    = h(<ci3, eier, '1'>)
179         ci5    = h(<ci4, sier, '1'>)
180         ci6    = h(<ci5, ~psk, '1'>)
```

```
181         hit    = h(<ci5, ~psk, '2'>)
182         ki6    = h(<ci5, ~psk, '3'>)
183         hi3    = h(<hi2, hit>)
184         aempt  = aead(ki6, 'e', hi3)
185         hi4    = h(<hi3, aempt>)
186         m2     = <'2', sidI, sidR, pekR, aempt, $mac1, $mac2>
187         /* We assume the first transport message is 'as weak as possible',
188          * that is, all values in it are public other than the key used to
189          * encrypt. */
190         mtr    = <'4', sidR, aead(ci6, 'message', '0')> in
191     [ State(~id,invar,<'init',sidI,~ekI,ci2,ki2,hi1>)[+]
192     , In(m2)
193     ]--[
194       /* In the real protocol, this isn't actually ci6, but is rather derived
195        * from it via some more hashing, but it doesn't make a difference here
196        * (as with cr6 above). */
197       IKeys(<pkI, pkR, pekI, pekR, ~psk, ci6>)
198     , Invariant(~id,invar)
199     ]->
200     [ State(~id,invar,<'transport',sidI,ci6>)
201     , Out(mtr)
202     ]
203
204 /* === Key Confirmation ===
205  *
206  * For proving agreement properties after the first transport message
207  * This is not faithfully modelled, since we don't care about the contents
208  * of the transport message--instead it's generically 'anything encrypted
209  * with the transport key I just set up'. */
210
211 rule R_ConfirmedTransportMessage:
212     let invar = <~psk,~ltkR,pkI,sisr>
213         mtr   = <'4', sid, data> in
214     [ State(~id,invar,<'transport_unconfirmed',sidR,pekI,pekR,cr>)[+]
215     , In(mtr)
216     ]--[
217       RConfirm(<pkI, 'g'^~ltkR, pekI, pekR, ~psk, cr>)
218       //We manually verify this instead of the recomputation used for other
219       //terms above, since morally R might not know the contents of the message.
220     , Eq(verify(data, '0', cr), true)
221     , Invariant(~id,invar)
222     ]->
223     [ State(~id,invar,<'transport',sidR,cr>) ]
224
225
226 /* === Session Traces ===
227  *
228  * We show that at least the protocol works. */
229
230 lemma exists_session: exists-trace
231     "Ex pki pkr peki pekr psk ck #i #j.
232         IKeys(<pki, pkr, peki, pekr, psk, ck>) @ i
233         & RConfirm(<pki, pkr, peki, pekr, psk, ck>) @ j & #i < #j"
234
235 /* There are only three numbers, 0, 1, and infinity. So 2 == infinity */
236 lemma exists_two_sessions: exists-trace
237     "Ex pki pkr sesskeys sesskeys2 #i #j #i2 #j2.
238         IKeys(<pki, pkr, sesskeys>) @ i
239         & RConfirm(<pki, pkr, sesskeys>) @ j & #i < #j
240         & IKeys(<pki, pkr, sesskeys2>) @ i2
241         & RConfirm(<pki, pkr, sesskeys2>) @ j2 & #i2 < #j2
242         & #i < #i2 & not(sesskeys = sesskeys2)"
243
244 // === Invariant Source Lemma ===
245 lemma invariants[sources]:
246     "All id invar #i.
247         Invariant(id,invar) @ i ==> Ex #j. #j < #i & InvariantSource(id,invar) @ j"
248
249 /* === Agreement Properties === */
250 lemma I_disagreement_implies_Sr_or_SiEi_compromise_and_PSK_compromise[reuse]:
251     "All pki pkr peki pekr psk ck #i.
252         /* If I believes they have completed a handshake with R */
253         IKeys(<pki, pkr, peki, pekr, psk, ck>) @ i
254         /* but R doesn't have a matching session */
255         & not(Ex #j. #j < #i & RKeys(<pki, pkr, peki, pekr, psk, ck>) @ j)
256     ==> /* Then the PSK was compromised (or not in use) and */
257         (Ex #j. Reveal_PSK(psk) @ j & #j < #i) & (
258         /* either R's static was compromised */
259         (Ex #j. Reveal_AK(pkr) @ j & #j < #i)
260         /* or both I's static and ephemeral were already compromised */
261         | (Ex #j #j2. Reveal_AK(pki) @ j & #j < #i
262             & Reveal_EphK(peki) @ j2 & #j2 < #i)
263         )"
264
265 lemma R_disagreement_implies_Si_or_SrEr_compromise_and_PSK_compromise[reuse]:
266     "All pki pkr peki pekr psk ck #i.
```

```
267 |         /* If R believes they have a confirmed session with I */
268 |         RConfirm(<pki, pkr, peki, pekr, psk, ck>) @ i
269 |         /* but I doesn't have a matching session */
270 |         & not(Ex #j. #j < #i & IKeys(<pki, pkr, peki, pekr, psk, ck>) @ j)
271 |     ==> /* Then the PSK was compromised (or not in use) and */
272 |         (Ex #j. Reveal_PSK(psk) @ j & #j < #i) & (
273 |         /* either I's static was compromised */
274 |         (Ex #j. Reveal_AK(pki) @ j & #j < #i)
275 |         /* or both R's static and ephemeral were already compromised */
276 |         | (Ex #j #j2. Reveal_AK(pkr) @ j & Reveal_EphK(pekr) @ j2
277 |             & #j2 < #i & #j < #i)
278 |         )"
279 |
280 | lemma UKS_resistance:
281 |     "All pki1 pki2 pkr1 pkr2 peki1 peki2 pekr1 pekr2 psk1 psk2 ck #i #j.
282 |         IKeys(<pki1, pkr1, peki1, pekr1, psk1, ck>) @ i
283 |         & RKeys(<pki2, pkr2, peki2, pekr2, psk2, ck>) @ j
284 |     ==> pki1 = pki2 & pkr1 = pkr2 & peki1 = peki2 & pekr1 = pekr2 & psk1 = psk2"
285 |
286 | lemma session_uniqueness:
287 |     /* For both I and R, a 'confirmed session key' will always be unique
288 |      * (even if all keys are compromised). This follows from I and R generating
289 |      * random ephemeral values. Note that this could be violated if the
290 |      * RNG can be controlled instead of just revealed, which we do not model. */
291 |     "(All pki pkr peki pekr psk ck #i.
292 |         IKeys(<pki, pkr, peki, pekr, psk, ck>) @ i
293 |     ==> not(Ex peki2 pekr2 #k.
294 |             IKeys(<pki, pkr, peki2, pekr2, psk, ck>) @ k & not(#k = #i)))
295 |     &(All pki pkr peki pekr psk ck #i.
296 |         RConfirm(<pki, pkr, peki, pekr, psk, ck>) @ i
297 |     ==> not(Ex peki2 pekr2 psk2 #k.
298 |             RConfirm(<pki, pkr, peki2, pekr2, psk2, ck>) @ k & not(#k = #i)))"
299 |
300 | /* === Secrecy Properties === */
301 |
302 | lemma key_secrecy[reuse]:
303 |     "(All pki pkr peki pekr psk ck #i #i2.
304 |         /* If I and R agree on keys */
305 |         IKeys(<pki, pkr, peki, pekr, psk, ck>) @ i
306 |         & RKeys(<pki, pkr, peki, pekr, psk, ck>) @ i2
307 |     ==> /* Either the adversary doesn't know the keys */
308 |         not(Ex #j. K(ck) @ j)
309 |         /* or the psk was compromised (or not in use) */
310 |         | ((Ex #j. Reveal_PSK(psk) @ j) & (
311 |            /* and pair of keys from the same agent was revealed (at any time) */
312 |            (Ex #j #j2. Reveal_AK(pki) @ j & Reveal_EphK(peki) @ j2)
313 |          | (Ex #j #j2. Reveal_AK(pkr) @ j & Reveal_EphK(pekr) @ j2))
314 |         ))"
315 |
316 | lemma identity_hiding:
317 |     /* Since the public static is always symbolically known to the adversary,
318 |      * we represent identity hiding by whether the adversary could learn some
319 |      * other value that is put in the first message at the
320 |      * same position as the public static. */
321 |     "All pki pkr peki pekr ck surrogate #i #j.
322 |         /* If R received a handshake init with a particular identity surrogate */
323 |         RKeys(<pki, pkr, peki, pekr, ck>) @ i & Identity_Surrogate(surrogate) @ i
324 |         /* And the adversary knows that surrogate for the identity */
325 |         & K(surrogate) @ j
326 |     ==> /* Then adversary compromised at least one of
327 |          * R's static, I's static, or I's ephemeral */
328 |         (Ex #j. Reveal_AK(pkr) @ j)
329 |         | (Ex #j. Reveal_AK(pki) @ j)
330 |         | (Ex #j. Reveal_EphK(peki) @ j)"
331 |
332 | end
```

A.2 DNP3

```
1  |
2  |
3  | theory DNP3
4  | begin
5  | /*****************************************************
6  | * ==============================
7  | *  DNP3 Secure Authentication v5
8  | * ==============================
9  | *
10 | * Author: Martin Dehnel-Wild and Kevin Milner
```

```
* Date: Apr 2017
* Status: Complete
*
**********************************************************/

builtins: hashing, symmetric-encryption

functions: hmac/2

//restriction Eq_testing: "All x y #i. Eq( x, y ) @ i ==> x = y"

restriction InEq_testing: "All x y #i. InEq( x, y ) @ i ==> not( x = y )"
restriction Unique_Pairings_id:
    "All x #i #j. Unique( x ) @ i & Unique( x ) @ j ==> #i = #j"
// This enforces that x and y are of distinct 'types': specifically in this case,
//  no outstation ID will end up being used as a user ID and vice versa
restriction USR_and_OutstationID_distinct: "All x y #i. Distinct( x, y ) @ i
    ==> not( Ex #j z. Distinct( y, z ) @ j ) & not( Ex #j z. Distinct( z, x ) @ j )"

/*********************************************************
 * Adversary Rules
 *********************************************************/

rule Update_key_reveal:
    [ !UpdateKeyToReveal( ~linkid, k ) ]
  --[ UpdateKeyReveal( k ), AdversaryRule( 'Update_key_reveal' ) ]->
    [ Out( k ) ]

rule cdsk_reveal:
    [ !CDSKToReveal( k1 ) ]
  --[ CDSKReveal( k1), AdversaryRule( 'Update_key_reveal' ) ]->
    [ Out( k1 ) ]

rule mdsk_reveal:
    [ !MDSKToReveal( k1 ) ]
  --[ MDSKReveal( k1 ), AdversaryRule( 'Update_key_reveal' ) ]->
    [ Out( k1 ) ]

rule authority_key_reveal:
    [ !F_AuthorityKey(k1 ) ]
  --[ AuthorityKeyReveal( k1 ), AdversaryRule( 'Update_key_reveal' ) ]->
    [ Out( k1 ) ]

/*********************************************************
 * Auxiliary Rules
 *********************************************************/

// The Authority's single point of key generation.
rule Authority_Key:
    [ Fr( ~AK ) ]
  --[ ]->
    [ !F_AuthorityKey(~AK ) ]


// L_Counter() rule. We say that 1+1 = h(1) (e.g. successor function).
rule CountUp:
    [ L_Counter(~id,val )
        ]
  --[ NewCounterValue( ~id, h( val ) ) ]->
    [ L_Counter(~id,h( val ) )
    , L_CounterValue(~id,h( val ) )
    , Out( h( val ) ) // Just so we can be sure counters are public
        ]


/*********************************************************
 * Initial Setup
 *********************************************************/

rule Initial_key_pre_distribution:
    let invars = <~AK,$USR,$OUTSTATION,~linkid>
        skinvars = <~skid,'NOT_INIT', 'undefined', 'undefined'>
        ukinvars = <~ukid, ~UK>
        chal     = <~cid, ~skid, 'none'>
    in
    [ !F_AuthorityKey(~AK ) // Receive Authority Key securely.
    , Fr( ~UK )      // Generate first Update Key, between USR and Outstation
    , Fr( ~uid ), Fr( ~oid ) // State invariant identifiers
  , Fr(~cid) // Identifier for the challenge invariants
    , Fr( ~linkid )      // The unique link identifier between user and outstation
    , Fr(~skid), Fr(~ukid) //Identifier for session key and update key invariant
        ]
  --[ // The association of user number to outstation is unique *per authority key*.
      Unique( < ~AK, $USR, $OUTSTATION > )
    , Unique( < ~AK, $OUTSTATION, $USR > )
    , Distinct( $USR, $OUTSTATION )

```

```
98      , NewCounterValue( ~uid, '0' ), NewCounterValue( ~oid, '0' )
99      , NewUpdateKey( ~linkid, ~UK )
100
101     , U_InvariantSource(~uid, invars)
102     , O_InvariantSource(~oid, invars)
103     , InvariantSource1(<~uid,'uk'>,ukinvars),InvariantSource2(<~uid,'uk'>,ukinvars)
104     , InvariantSource1(<~oid,'uk'>,ukinvars),InvariantSource2(<~oid,'uk'>,ukinvars)
105     , InvariantSource1(<~uid,'sk'>,skinvars),InvariantSource2(<~uid,'sk'>,skinvars)
106     , InvariantSource1(<~oid,'sk'>,skinvars),InvariantSource2(<~oid,'sk'>,skinvars)
107     , InvariantSource1(<~uid,'c'>,chal),InvariantSource2(<~uid,'c'>,chal)
108     , InvariantSource1(<~oid,'c'>,chal),InvariantSource2(<~oid,'c'>,chal)
109     , InvariantSource1(<~uid,'m'>,chal),InvariantSource2(<~uid,'m'>,chal)
110     , InvariantSource1(<~oid,'m'>,chal),InvariantSource2(<~oid,'m'>,chal)
111     ]->
112     [ UserState( ~uid, invars,  ukinvars, skinvars,
113                   <'0', chal>, <'0', chal>, 'Init' )
114     , OutstationState( ~oid, invars, '0', ukinvars, skinvars,
115                   <'0', chal>, <'0', chal>, 'SecurityIdle' )
116
117     // This is the last key status message, which the outstation is expected
118     //  to accept key changes on, even if things have happened since it was sent
119     , OutSentKeyStatus( ~oid, invars, skinvars, 'none' )
120
121     // These are the counter facts, which will go into a special 'count up' rule
122     //  so we can easily prove monotonicity (and uniqueness) of counter values
123     , L_Counter(~uid,'0' ), L_Counter(~oid,'0' )
124
125     , !F_UpdateKey(~linkid,~UK )   // For e.g. adversary key-reveal events
126     ]
127
128 /*******************************************************************************
129  * Session Key Update Protocol
130  *******************************************************************************/
131
132 /*********************************************************
133  * Send `Session Key Status Request' (g120v4)
134  * Sent by User
135  *********************************************************/
136
137 rule S1_SKSR_session_key_status_request:
138     let invars = <~AK,$USR,$OUTSTATION,~linkid>
139     in
140     [ UserState( ~id, invars, ukinvars, skinvars,
141                   <cCSQ, cChal> , <mCSQ, mChal>, anystate )[-]
142         ]
143   --[ U_Invariant(~id, invars)
144     , L_Invariant(<~id, 'sk'>, skinvars)
145     , L_Invariant(<~id, 'uk'>, ukinvars)
146     , L_Invariant(<~id, 'c'>, cChal)
147     , L_Invariant(<~id, 'm'>, mChal)
148     ]->
149     [ UserState( ~id, invars, ukinvars, skinvars,
150                   <cCSQ, cChal> , <mCSQ, mChal>, 'SessionKeyChange' )
151     , Out( $USR )
152     ]
153
154 /*********************************************************
155  * Send `Session Key Status' Message (g120v5)
156  * Sent by Outstation
157  *********************************************************/
158
159 rule S2_SKS_session_key_status:
160     let invars = <~AK,$USR,$OSID,~linkid>
161         skinvars = <~skid,$keystatus, CDSK, MDSK>
162         SKSM_j = < h( KSQ ), $USR, $keystatus, ~CD_j >
163     in
164     [ OutstationState( ~id, invars, KSQ, ukinvars, skinvars,
165                       <cCSQ, cChal> , <mCSQ, mChal>, 'SecurityIdle' )[-]
166     , OutSentKeyStatus( ~id, invars, skinvars, lastkeystatus )[-]
167     , Fr( ~CD_j )
168     , In( $USR )
169         ]
170   --[ O_Invariant(~id, invars)
171     , Invariant(<~id,'sk'>,skinvars),L_Invariant(<~id,'sk'>,skinvars)
172     , L_Invariant(<~id, 'uk'>, ukinvars)
173     , L_Invariant(<~id, 'c'>, cChal)
174     , L_Invariant(<~id, 'm'>, mChal)
175     ]->
176     [ OutstationState( ~id, invars, h( KSQ ), ukinvars, skinvars,
177                       <cCSQ, cChal> , <mCSQ, mChal>, 'SecurityIdle' )
178     , OutSentKeyStatus( ~id,invars, skinvars,  SKSM_j )
179     , Out( SKSM_j )
180     ]
181
182 /*********************************************************
```

```
 * Send ‘Session Key Change’ Message (g120v6)
 * Sent by User
 ******************************************************/

rule S3_SKC_session_key_change:
    let invars = <~AK,$USR,$OSID,~linkid>
        ukinvars = <~ukid,UK>
        newskinvars = <~skid,'OK', ~CDSK, ~MDSK>
        SKSM_j = < KSQ, $USR, $keystatus, CD_j >
        SKCM_j = < KSQ, $USR,
             senc( < ~CDSK, ~MDSK, SKSM_j >, UK) > in
    [ UserState( ~id, invars, ukinvars, skinvars,
                   <cCSQ, cChal> , <mCSQ, mChal>, 'SessionKeyChange' )[+]
    , Fr( ~CDSK ), Fr( ~MDSK ) // The new session keys
    , Fr( ~skid )
    , In( SKSM_j )
        ]
  --[ NewSKs( ~linkid, UK, ~CDSK, ~MDSK )
    , Sourced_UpdateKey( ~linkid, UK )
    , UpdateKeyUsedForSKs( ~linkid, UK, ~CDSK, ~MDSK )

    , U_Invariant(~id, invars)
    , InvariantSource1(<~id,'sk'>,newskinvars)
    , InvariantSource2(<~id,'sk'>,newskinvars)
    , Invariant(<~id,'uk'>,ukinvars),L_Invariant(<~id,'uk'>,ukinvars)
    , L_Invariant(<~id, 'c'>, cChal)
    , L_Invariant(<~id, 'm'>, mChal)
    ]->
    [ UserState( ~id, invars, ukinvars, newskinvars, <cCSQ, cChal> , <mCSQ, mChal>,
                   < 'WaitForKeyChangeConfirmation', SKCM_j, ~CDSK, ~MDSK > )
    , !CDSKToReveal( ~CDSK )
    , !MDSKToReveal( ~MDSK )
    , Out( SKCM_j )
    ]

/******************************************************
 * Send another ‘Session Key Status’ Message’ (g120v5)
 * Sent by Outstation
 ******************************************************/

rule S4_SKS_session_key_status:
    let invars = <~AK,$USR,$OSID,~linkid>
        ukinvars = <~ukid,UK>
        newskinvars = <~skid,'OK', CDSK, MDSK>
        newChal = <~cid, ~skid, 'none'>
        SKSM_j =  < KSQ, $USR, $keystatus, CD_j >
        SKCM_j =  < KSQ, $USR, senc( < CDSK, MDSK, SKSM_j >, UK ) >
        SKSM_j_plus_1 = < h( KSQ ), $USR, 'OK', ~CD_j_plus_1,
                           hmac( SKCM_j, MDSK ) >
    in
    [ OutstationState( ~id, invars, KSQ, ukinvars, skinvars,
                        < cCSQ, cChal >, < mCSQ, cChal >, 'SecurityIdle' )[-]
    , L_CounterValue(~id,h( cCSQ ) )
    , OutSentKeyStatus( ~id, invars, skinvars, SKSM_j )[-]
    , Fr( ~CD_j_plus_1 )
  , Fr(~cid)
    , Fr(~skid)
    , In( SKCM_j )
        ]
  --[ CSQ( ~id, h( cCSQ ) )
    , Sourced_UpdateKey( ~linkid, UK )
    , Sourced_SKs( ~linkid, UK, CDSK, MDSK )
    , UpdateKeyUsedForSKs( ~linkid, UK, CDSK, MDSK )

    , O_Invariant(~id, invars)
    , InvariantSource1(<~id,'sk'>,newskinvars)
    , InvariantSource2(<~id,'sk'>,newskinvars)
    , Invariant(<~id,'uk'>,ukinvars),L_Invariant(<~id,'uk'>,ukinvars)
    , InvariantSource1(<~id,'c'>,newChal),InvariantSource2(<~id,'c'>,newChal)
    , InvariantSource1(<~id,'m'>,newChal),InvariantSource2(<~id,'m'>,newChal)
    ]->
    [ // Drop last challenge on key update
      OutstationState( ~id, invars, h( KSQ ), ukinvars, newskinvars,
                        <h(cCSQ), newChal> , <h(mCSQ), newChal>, 'SecurityIdle' )
    , OutSentKeyStatus( ~id, invars, newskinvars, SKSM_j_plus_1 )
    , Out( SKSM_j_plus_1 )
    ]

/******************************************************
 * User receive SKS confirmation message from Outstation
 ******************************************************/

rule S5_receive_SKS_confirmation:
    let invars = <~AK,$USR,$OSID,~linkid>
        ukinvars = <~ukid,UK>
        skinvars = <~skid,'OK', CDSK, MDSK>
```

```
269          newChal = <~cid, ~skid, 'none'>
270          SKSM_j_plus_1 = < KSQ, $USR, 'OK', CD_j_plus_1,
271                                hmac( SKCM_j, MDSK ) >
272      in
273      [ UserState( ~id, invars, ukinvars, skinvars, < cCSQ, cChal >, < mCSQ, cChal >,
274                     < 'WaitForKeyChangeConfirmation', SKCM_j, CDSK, MDSK > )[+]
275      , L_CounterValue(~id,h( mCSQ ) )
276      , Fr( ~cid )
277      , In( SKSM_j_plus_1 )
278      ]
279    --[ CSQ( ~id, h( mCSQ ) )
280
281      , U_Invariant(~id, invars)
282      , L_Invariant(<~id, 'sk'>, skinvars)
283      , L_Invariant(<~id, 'uk'>, ukinvars)
284      , InvariantSource1(<~id,'c'>,newChal),InvariantSource2(<~id,'c'>,newChal)
285      , InvariantSource1(<~id,'m'>,newChal),InvariantSource2(<~id,'m'>,newChal)
286      ]->
287      [
288        UserState( ~id, invars, ukinvars, skinvars,
289                      <h(cCSQ), newChal> , <h(mCSQ), newChal>, 'SecurityIdle' )
290      ]
291 /*****************************************************************************
292  * Critical ASDU functionality
293  *****************************************************************************/
294
295 rule A2_C_AC_Authentication_Challenge: // g120v1
296      let invars = <AK,$USR,$OSID,~linkid>
297          AC = < h( cCSQ ), $USR, ~CD >
298      in
299      [ OutstationState( ~id, invars, KSQ, ukinvars, skinvars,
300                           <cCSQ, cChal>, <mCSQ, mChal>, 'SecurityIdle')[-]
301      , L_CounterValue(~id,h( cCSQ ) )
302      , Fr( ~CD )
303          ]
304    --[ CSQ( ~id, h( cCSQ ) )
305
306      , O_Invariant(~id, invars)
307      , L_Invariant(<~id,'sk'>, skinvars)
308      , L_Invariant(<~id,'uk'>, ukinvars)
309      , L_Invariant(<~id,'c'>, cChal)
310      , L_Invariant(<~id,'m'>, mChal)
311      ]->
312      [ OutstationState( ~id, invars, KSQ, ukinvars, skinvars,
313                           <cCSQ, cChal>, <mCSQ, mChal>,
314                           < 'WaitForReply', < h( cCSQ ), AC > > )
315      , Out( AC )
316      ]
317
318 rule A3_C_AR_Authentication_Reply: // g120v2
319      let invars = <AK,$USR,$OSID,~linkid>
320          skinvars = <~skid, 'OK', CDSK, MDSK>
321          AC = < CSQ, $USR, CD >
322          AR = < CSQ, $USR, hmac( < CSQ, AC, $ASDU >, CDSK ) >
323          newcChal = <~ccid, ~skid, AC>
324      in
325      [ UserState( ~id, invars, ukinvars, skinvars,
326                      <cCSQ, cChal>, <mCSQ, mChal>, 'SecurityIdle' )[-]
327    , Fr(~ccid)
328      , In( AC )
329      ]
330    --[ SentASDU( ~linkid, AR, 'normal', 'control' )
331      , UsingSessKeys( CDSK, MDSK )
332      , AuthReply( AC, $ASDU, CDSK )
333
334      , U_Invariant(~id, invars)
335      , Invariant(<~id,'sk'>,skinvars),L_Invariant(<~id,'sk'>,skinvars)
336      , L_Invariant(<~id,'uk'>, ukinvars)
337      , InvariantSource1(<~id,'c'>,newcChal),InvariantSource2(<~id,'c'>,newcChal)
338      , L_Invariant(<~id,'m'>, mChal)
339      ]->
340      [
341        UserState( ~id, invars, ukinvars, skinvars,
342                      < CSQ, newcChal >, <mCSQ, mChal>, 'SecurityIdle' )
343      , Out( AR )
344      ]
345
346 rule A3_C_AR_Authentication_Aggressive:
347      let invars = <AK,$USR,$OSID,~linkid>
348          skinvars = <~skid, 'OK', CDSK, MDSK>
349      cChal = <~ccid, ~skid, AC>
350          AR = < h( cCSQ ), $USR, hmac( < 'amode', h( cCSQ ), AC, $ASDU >, CDSK ) >
351      in
352      [ UserState( ~id, invars, ukinvars, skinvars,
353                      <cCSQ, cChal>, <mCSQ, mChal>, 'SecurityIdle' )[-]
```

```
354          ]
355   --[ SentASDU( ~linkid, AR, 'aggressive', 'control' )
356      , UsingSessKeys( CDSK, MDSK )
357      , AuthReply( cChal, $ASDU, CDSK )
358
359      , U_Invariant(~id, invars)
360      , Invariant(<~id,'sk'>,skinvars),L_Invariant(<~id,'sk'>,skinvars)
361      , L_Invariant(<~id,'uk'>, ukinvars)
362      , Invariant(<~id,'c'>,cChal),L_Invariant(<~id,'c'>,cChal)
363      , L_Invariant(<~id,'m'>, mChal)
364      ]->
365      [ UserState( ~id, invars, ukinvars, skinvars,
366                    < h( cCSQ ), cChal >, <mCSQ, mChal>, 'SecurityIdle' )
367      , Out( AR )
368      ]
369
370
371 // Outstation receives MAC'd value of the ASDU (non-agressive mode)
372 rule A4_receive_C_AC_of_ASDU:
373     let invars = <AK, $USR, $OSID, ~linkid>
374     skinvars = <~skid, 'OK', CDSK, MDSK>
375     AC = < CSQ, $USR, CD >
376         AR = < CSQ, $USR, hmac( < CSQ, AC, $ASDU >, CDSK ) >
377         newcChal = <~newccid, ~skid, AC>
378     in
379     [ OutstationState( ~id, invars, KSQ, ukinvars, skinvars,
380                        <cCSQ, cChal>, <mCSQ, mChal>,
381                        < 'WaitForReply', < CSQ, AC> > )[+]
382     , Fr(~newccid)
383     , In( AR )
384     ]
385   --[ AuthASDU( ~linkid, AR, 'normal', 'control' )
386     , UsingSessKeys( CDSK, MDSK )
387
388     , O_Invariant(~id, invars)
389     , Invariant(<~id,'sk'>,skinvars),L_Invariant(<~id,'sk'>,skinvars)
390     , L_Invariant(<~id,'uk'>, ukinvars)
391     , InvariantSource1(<~id,'c'>,newcChal),InvariantSource2(<~id,'c'>,newcChal)
392     , L_Invariant(<~id,'m'>, mChal)
393     ]->
394     [ OutstationState( ~id, invars, KSQ, ukinvars, skinvars,
395                        < CSQ, newcChal >, <mCSQ, mChal>, 'SecurityIdle' )
396     ]
397
398 // Aggressive mode, can receive in either WaitForReply or SecurityIdle,
399 // where waitforreply drops the previous message.
400 // In security idle, we need to get the next counter value to check the received CSQ
401 rule A4_idle_receive_C_AC_aggressive:
402     let invars = <AK, $USR, $OSID, ~linkid>
403     skinvars = <~skid, 'OK', CDSK, MDSK>
404     cChal   = <~ccid, ~skid, AC>
405     AR = < h( CSQ ), $USR, hmac( < 'amode', h( CSQ ), AC, $ASDU >, CDSK ) >
406     in
407     [ OutstationState( ~id, invars, KSQ, ukinvars, skinvars,
408                        < CSQ, cChal >, <mCSQ, mChal>, 'SecurityIdle' )[-]
409     , L_CounterValue(~id,h( CSQ ) )
410     , In( AR )
411     ]
412   --[ AuthASDU( ~linkid, AR, 'aggressive', 'control' )
413     , UsingSessKeys( CDSK, MDSK )
414     , CSQ( ~id, h( CSQ ) )
415     , InEq( AC, 'none' )
416
417     , O_Invariant(~id, invars)
418     , Invariant(<~id,'sk'>,skinvars),L_Invariant(<~id,'sk'>,skinvars)
419     , L_Invariant(<~id,'uk'>, ukinvars)
420     , Invariant(<~id,'c'>,cChal),L_Invariant(<~id,'c'>,cChal)
421     , L_Invariant(<~id,'m'>, mChal)
422     ]->
423     [ OutstationState( ~id, invars, KSQ, ukinvars, skinvars,
424                        < h( CSQ ), cChal >, <mCSQ, mChal>, 'SecurityIdle' )
425     ]
426
427 rule A4_waiting_receive_C_AC_aggressive:
428     let invars = <AK, $USR, $OSID, ~linkid>
429     skinvars = <~skid, 'OK', CDSK, MDSK>
430     cChal   = <~ccid, ~skid, AC>
431     AC = < h( CSQ ), $USR, CD >
432         AR = < h( CSQ ), $USR, hmac( < 'amode', h( CSQ ), AC, $ASDU >, CDSK ) >
433     in
434     [ OutstationState( ~id, invars, KSQ, ukinvars, skinvars,
435                        < CSQ, cChal >, <mCSQ, mChal>,
436                        < 'WaitForReply', <chalCSQ, chalAC> > )[+]
437     , In( AR )
438     ]
439   --[ AuthASDU( ~linkid, AR, 'aggressive', 'control' )
```

```
440       , UsingSessKeys( CDSK, MDSK )
441       , InEq( AC, 'none' )
442 
443       , O_Invariant(~id, invars)
444       , Invariant(<~id,'sk'>,skinvars),L_Invariant(<~id,'sk'>,skinvars)
445       , L_Invariant(<~id,'uk'>, ukinvars)
446       , Invariant(<~id,'c'>,cChal),L_Invariant(<~id,'c'>,cChal)
447       , L_Invariant(<~id,'m'>, mChal)
448       ]->
449       [ OutstationState( ~id, invars, KSQ, ukinvars, skinvars,
450                          < h( CSQ ), cChal >, <mCSQ, mChal>, 'SecurityIdle' )
451       ]
452 
453 // When timing or erroring out, we still replace the last
454 // sent challenge with the more recent one.
455 rule A_OutstationWaitForReply_TimeoutorError:
456       let skinvars = <~skid, 'OK', CDSK, MDSK>
457           newcChal = <~newccid, ~skid, AC>
458       in
459       [ OutstationState( ~id, invars, KSQ, ukinvars, skinvars,
460                          < CSQ, cChal >, <mCSQ, mChal>,
461                          < 'WaitForReply', <chalCSQ, AC> > )[+]
462       , Fr(~newccid)
463     ]
464     --[ O_Invariant(~id, invars)
465       , L_Invariant(<~id,'sk'>, skinvars)
466       , L_Invariant(<~id,'uk'>, ukinvars)
467       , InvariantSource1(<~id,'c'>,newcChal),InvariantSource2(<~id,'c'>,newcChal)
468       , L_Invariant(<~id,'m'>, mChal)
469       ]->
470       [ OutstationState( ~id, invars, KSQ, ukinvars, skinvars,
471                          <chalCSQ,newcChal>, <mCSQ, mChal>, 'SecurityIdle' )
472       ]
473 
474 /*************************************************
475  * Critical ASDU Protocol, Monitoring direction
476  *************************************************/
477 
478 rule A2_M_AC_Authentication_Challenge: // g120v1
479       let invars = <AK, $USR, $OSID, ~linkid>
480       AC = < h( mCSQ ), $USR, ~CD >
481     in
482       [ UserState( ~id, invars, ukinvars, skinvars,
483                    <cCSQ, cChal>, < mCSQ, mChal >, 'SecurityIdle' )[-]
484       , L_CounterValue(~id,h( mCSQ ) )
485       , Fr( ~CD )
486     , Fr(~mcid)
487       ]
488     --[ CSQ( ~id, h( mCSQ ) )
489 
490     , U_Invariant(~id, invars)
491       , L_Invariant(<~id,'sk'>, skinvars)
492       , L_Invariant(<~id,'uk'>, ukinvars)
493       , L_Invariant(<~id,'c'>, cChal)
494       , L_Invariant(<~id,'m'>, mChal)
495       ]->
496       [ UserState( ~id, invars, ukinvars, skinvars,
497                    <cCSQ, cChal>, < mCSQ, mChal >,
498                    < 'WaitForReply', < h( mCSQ ), AC > > )
499       , Out( AC )
500       ]
501 
502 
503 rule A3_M_AR_Authentication_Reply:
504       let invars = <AK, $USR, $OSID, ~linkid>
505       skinvars = <~skid, 'OK', CDSK, MDSK>
506       AC = < CSQ, $USR, CD >
507           AR = < CSQ, $USR, hmac(< CSQ, AC, $ASDU >, MDSK ) >
508       newmChal = <~mcid, ~skid, AC>
509       in
510       [ OutstationState( ~id, invars, KSQ, ukinvars, skinvars,
511                          <cCSQ, cChal>, <mCSQ, mChal>, 'SecurityIdle')[-]
512       , Fr( ~mcid )
513       , In( AC )
514       ]
515     --[ SentASDU( ~linkid, AR, 'normal', 'monitor')
516       , UsingSessKeys( CDSK, MDSK )
517       , AuthReply( AC, $ASDU, MDSK )
518 
519     , O_Invariant(~id, invars)
520       , Invariant(<~id,'sk'>,skinvars),L_Invariant(<~id,'sk'>,skinvars)
521       , L_Invariant(<~id,'uk'>, ukinvars)
522       , L_Invariant(<~id,'c'>, cChal)
523       , InvariantSource1(<~id,'m'>,newmChal)
524       , InvariantSource2(<~id,'m'>,newmChal)
525       ]->
```

```
526      [ OutstationState( ~id, invars, KSQ, ukinvars, skinvars,
527                                <cCSQ, cChal>, < CSQ, newmChal>, 'SecurityIdle' )
528      , Out( AR )
529      ]
530
531  rule A3_M_AR_Authentication_Aggressive:
532      let invars = <AK, $USR, $OSID, ~linkid>
533      skinvars = <~skid, 'OK', CDSK, MDSK>
534      mChal = <~mcid, ~skid, AC>
535      AR = < h( mCSQ ), $USR, hmac( < 'amode', h( mCSQ ), AC, $ASDU >, MDSK ) >
536      in
537      [ OutstationState( ~id, invars, KSQ, ukinvars, skinvars,
538                                <cCSQ, cChal>, < mCSQ, mChal >, 'SecurityIdle' )[-]
539      ]
540    --[ SentASDU( ~linkid, AR, 'aggressive', 'monitor' )
541      , UsingSessKeys( CDSK, MDSK )
542      , AuthReply( mChal, $ASDU, MDSK )
543
544    , O_Invariant(~id, invars)
545      , Invariant(<~id,'sk'>,skinvars),L_Invariant(<~id,'sk'>,skinvars)
546      , L_Invariant(<~id,'uk'>, ukinvars)
547      , L_Invariant(<~id,'c'>, cChal)
548      , Invariant(<~id,'m'>,mChal),L_Invariant(<~id,'m'>,mChal)
549      ]->
550      [ OutstationState( ~id, invars, KSQ, ukinvars, skinvars,
551                                <cCSQ, cChal>, < h( mCSQ ), mChal >, 'SecurityIdle' )
552      , Out( AR )
553      ]
554
555
556  rule A4_receive_M_AC_of_ASDU:
557      let invars = <AK, $USR, $OSID, ~linkid>
558      skinvars = <~skid,'OK', CDSK, MDSK>
559      AC = < CSQ, $USR, CD >
560          AR = < CSQ, $USR, hmac( < CSQ, AC, $ASDU >, MDSK ) >
561          newmChal = <~newmcid, ~skid, AC>
562      in
563      [ UserState( ~id, invars, ukinvars, skinvars,
564                          <cCSQ, cChal>, < mCSQ, mChal >,
565                          < 'WaitForReply', < CSQ, AC > > )[+]
566      , Fr(~newmcid)
567      , In( AR )
568      ]
569    --[ AuthASDU( ~linkid, AR, 'normal', 'monitor' )
570      , UsingSessKeys( CDSK, MDSK )
571
572    , U_Invariant(~id, invars)
573      , Invariant(<~id,'sk'>,skinvars),L_Invariant(<~id,'sk'>,skinvars)
574      , L_Invariant(<~id,'uk'>, ukinvars)
575      , L_Invariant(<~id,'c'>, cChal)
576      , InvariantSource1(<~id,'m'>,newmChal)
577      , InvariantSource2(<~id,'m'>,newmChal)
578      ]->
579      [ UserState( ~id, invars, ukinvars, skinvars,
580                          <cCSQ, cChal>, < CSQ, newmChal >, 'SecurityIdle' )
581      ]
582
583  rule A4_idle_receive_M_AC_aggressive:
584      let invars = <AK, $USR, $OSID, ~linkid>
585      skinvars = <~skid, 'OK', CDSK, MDSK>
586      mChal = <~mcid, ~skid, AC>
587      AR = < h( mCSQ ), $USR, hmac( < 'amode', h( mCSQ ), AC, $ASDU >, MDSK ) >
588      in
589      [ UserState( ~id, invars, ukinvars, skinvars,
590                          <cCSQ, cChal>, < mCSQ, mChal >, 'SecurityIdle' )[-]
591      , L_CounterValue(~id,h( mCSQ ) )
592      , In( AR )
593      ]
594    --[ AuthASDU( ~linkid, AR, 'aggressive', 'monitor' )
595      , UsingSessKeys( CDSK, MDSK )
596      , CSQ( ~id, h( mCSQ ) )
597      , InEq( AC, 'none' )
598
599    , U_Invariant(~id, invars)
600      , Invariant(<~id,'sk'>,skinvars),L_Invariant(<~id,'sk'>,skinvars)
601      , L_Invariant(<~id,'uk'>, ukinvars)
602      , L_Invariant(<~id,'c'>, cChal)
603      , Invariant(<~id,'m'>,mChal),L_Invariant(<~id,'m'>,mChal)
604      ]->
605      [ UserState( ~id, invars, ukinvars, skinvars,
606                          <cCSQ, cChal>, < h( mCSQ ), mChal >, 'SecurityIdle' )
607      ]
608
609  rule A4_waiting_receive_M_AC_aggressive:
610      let invars = <AK, $USR, $OSID, ~linkid>
611      skinvars = <~skid, 'OK', CDSK, MDSK>
```

```
612           mChal = <~mcid, ~skid, AC>
613       AR = < h( mCSQ ), $USR, hmac( < 'amode', h( mCSQ ), AC, $ASDU >, MDSK ) >
614       in
615       [ UserState( ~id, invars, ukinvars, skinvars,
616                        <cCSQ, cChal>, < mCSQ, mChal >,
617                        < 'WaitForReply', <CSQChal,ACChal  > > )[+]
618       , In( AR )
619       ]
620     --[ AuthASDU( ~linkid, AR, 'aggressive', 'monitor' )
621       , UsingSessKeys( CDSK, MDSK )
622       , InEq( AC, 'none' )
623 
624     , U_Invariant(~id, invars)
625       , Invariant(<~id,'sk'>,skinvars),L_Invariant(<~id,'sk'>,skinvars)
626       , L_Invariant(<~id,'uk'>, ukinvars)
627       , L_Invariant(<~id,'c'>, cChal)
628       , Invariant(<~id,'m'>,mChal),L_Invariant(<~id,'m'>,mChal)
629       ]->
630       [ UserState( ~id, invars, ukinvars, skinvars,
631                        <cCSQ, cChal>, < h( mCSQ ), mChal >, 'SecurityIdle' )
632       ]
633 
634 rule A_UserWaitForReply_Timeout:
635       let skinvars = <~skid, 'OK', CDSK, MDSK>
636           newmChal = <~newmcid, ~skid, AC>
637       in
638       [ UserState( ~id, invars, ukinvars, skinvars,
639                        <cCSQ, cChal>, <mCSQ, mChal>,
640                        < 'WaitForReply', <chalCSQ, AC> > )[+]
641       , Fr(~newmcid)
642       ]
643     --[ U_Invariant(~id, invars)
644       , L_Invariant(<~id,'sk'>, skinvars)
645       , L_Invariant(<~id,'uk'>, ukinvars)
646       , L_Invariant(<~id,'c'>, cChal)
647       , InvariantSource1(<~id,'m'>,newmChal)
648       , InvariantSource2(<~id,'m'>,newmChal)
649      ]->
650       [ UserState( ~id, invars, ukinvars, skinvars,
651                        <cCSQ, cChal>, <chalCSQ, newmChal>, 'SecurityIdle' )
652       ]
653 
654 /**********************************************************************************
655  * Update Key Change protocol
656  *********************************************************************************/
657 
658 rule U2_UKCRp_Key_Change_Reply:
659       [ Fr( ~CD_b ) ]
660     --[]->
661     [ OutUpdateKeyChallenge( $USR, ~CD_b )
662       , Out( < $USR, ~CD_b > )
663       ]
664 
665 rule U3_U4_U5_new_update_key:
666       let invars = <~AK, $USR, $OSID, ~linkid>
667       UKCRp = < KSQ, $USR, CD_b >
668           UKC   = < KSQ, $USR, senc( < $USR, ~UK, CD_b >, ~AK ) >
669           UKCCu = hmac( < $OSID, ~CD_a, CD_b, KSQ >, ~UK )
670       in
671       [ UserState( ~id, invars, ukinvars, skinvars,
672                        <cCSQ, cChal>, <mCSQ, mChal>, 'SecurityIdle' )[-]
673       , Fr( ~CD_a )
674       , Fr( ~UK )
675       , In( UKCRp )
676       ]
677     --[ NewUpdateKey( ~linkid, ~UK )
678 
679     , U_Invariant(~id, invars)
680       , L_Invariant(<~id,'sk'>, skinvars)
681       , L_Invariant(<~id,'uk'>, ukinvars)
682       , L_Invariant(<~id,'c'>, cChal)
683       , L_Invariant(<~id,'m'>, mChal)
684       ]->
685       [ UserState( ~id, invars, ukinvars, skinvars,
686                        <cCSQ, cChal>, <mCSQ, mChal>, < 'WaitForKCC', UKCCu > )
687       , !UpdateKeyToReveal( ~linkid, ~UK )
688       , Out( < ~CD_a, UKC, UKCCu > )
689       ]
690 
691 
692 // Since we commuted the KSQ update to U6, we should expect h(KSQ) as input
693 rule U6_UKCC_Update_Key_Change_Confirmation:
694       let invars = <AK, $USR, $OSID, ~linkid>
695       UKC = < h( KSQ ), $USR, senc( < $USR, UK, CD_b >, AK ) >
696           UKCCu = hmac( < $OSID, CD_a, CD_b, h( KSQ ) >, UK )
697           UKCCo = hmac( < $USR, CD_b, CD_a, h( KSQ ) >, UK )
```

```
698      newukinvars = <~newuid,UK>
699      in
700      [ OutstationState( ~id, invars, KSQ, ukinvars, skinvars,
701                         <cCSQ, cChal>, <mCSQ, mChal>, 'SecurityIdle' )[-]
702      , OutUpdateKeyChallenge( $USR, CD_b )[+]
703      , Fr( ~newuid )
704      , In( CD_a )
705      , In( < UKC, UKCCu > )
706      ]
707    --[ Sourced_UpdateKey( ~linkid, UK )
708
709    , O_Invariant(~id, invars)
710      , L_Invariant(<~id,'sk'>, skinvars)
711      , InvariantSource1(<~id,'uk'>,newukinvars)
712      , InvariantSource2(<~id,'uk'>,newukinvars)
713      , L_Invariant(<~id,'c'>, cChal)
714      , L_Invariant(<~id,'m'>, mChal)
715      ]->
716      [ OutstationState( ~id, invars, h( KSQ ), newukinvars, skinvars,
717                         <cCSQ,cChal>, <mCSQ,mChal>, 'SecurityIdle' )
718      , Out( UKCCo )
719      ]
720
721  rule U7_receive_UKCC_from_Outstation:
722      let invars = <AK, $USR, $OSID, ~linkid>
723      UKCCu = hmac( < $OSID, CD_a, CD_b, KSQ >, UK )
724          UKCCo = hmac( < $USR, CD_b, CD_a, KSQ >, UK )
725      newukinvars = <~newuid, UK>
726      in
727      [ UserState( ~id, invars, ukinvars, skinvars,
728                   <cCSQ, cChal>, <mCSQ, mChal>, < 'WaitForKCC', UKCCu > )[+]
729      , Fr( ~newuid )
730      , In( UKCCo )
731          ]
732    --[ Sourced_UpdateKey( ~linkid, UK )
733
734
735    , U_Invariant(~id, invars)
736      , L_Invariant(<~id,'sk'>, skinvars)
737      , InvariantSource1(<~id,'uk'>,newukinvars)
738      , InvariantSource2(<~id,'uk'>,newukinvars)
739      , L_Invariant(<~id,'c'>, cChal)
740      , L_Invariant(<~id,'m'>, mChal)
741      ]->
742      [ UserState( ~id, invars, newukinvars, skinvars,
743                   <cCSQ, cChal>, <mCSQ, mChal>, 'SecurityIdle' )
744      ]
745
746
747  /*********************************************************
748  * Lemmas from here on in. These are the things we prove.
749  *********************************************************/
750
751  // The U and O invariants have unique sources, so we separate them
752  lemma u_invariant_sources[sources]:
753    "All id invar #i.
754      U_Invariant(id, invar) @ i ==> Ex #j. #j < #i & U_InvariantSource(id, invar) @ j"
755
756  lemma o_invariant_sources[sources,heuristic=p]:
757    "All id invar #i.
758      O_Invariant(id, invar) @ i ==> Ex #j. #j < #i & O_InvariantSource(id, invar) @ j"
759
760  lemma invariant_sources[sources,heuristic=p]:
761      "(All id invar #i.
762          Invariant(id, invar) @ i
763          ==> Ex #j. #j < #i & InvariantSource1(id, invar) @ j)
764      &(All id invar #i.
765          L_Invariant(id, invar) @ i
766          ==> Ex #j. #j < #i & InvariantSource1(id,invar)[-] @ j)"
767
768  lemma invariant_sources_unique[reuse]:
769      "All id invar #i #j.
770          InvariantSource1(id, invar) @ i & InvariantSource2(id, invar) @ j
771      ==> #i = #j"
772
773  lemma countervalue_uniqueness[reuse, use_induction]:
774      "All id x #i #j.
775          NewCounterValue( id, x ) @ i & NewCounterValue( id, x ) @ j ==> #i = #j"
776
777  lemma CSQ_Uniqueness[reuse, use_induction]:
778      "All id csq #i #j.
779          CSQ( id, csq ) @ i & CSQ( id, csq ) @ j ==> #i = #j"
780
781  lemma authed_sessions_unique[reuse]:
782      "( All id ar mode mode2 direction #i #j.
783          AuthASDU( id, ar, mode, direction ) @ i
```

```
784        & AuthASDU( id, ar, mode2, direction ) @ j ==> #i = #j )"
785
786  lemma update_key_sourced[reuse, use_induction]:
787      "not( Ex ak #r. AuthorityKeyReveal( ak ) @ r)
788      ==>
789      ( All id uk #i.
790          not( Ex #r. UpdateKeyReveal( uk ) @ r & #r < #i )
791          & Sourced_UpdateKey( id,uk ) @ i
792          ==> Ex #j. #j < #i & NewUpdateKey( id, uk ) @ j)"
793
794  lemma update_key_secrecy:
795      "not( Ex ak #r. AuthorityKeyReveal( ak ) @ r )
796      ==>
797      ( All id uk #i.
798          not( Ex #r. UpdateKeyReveal( uk ) @ r )
799          & NewUpdateKey( id, uk ) @ #i
800          ==> not( Ex #j. K( uk ) @ #j))"
801
802  lemma sessionkey_secrecy_outst:
803      "not(Ex ak #r. AuthorityKeyReveal( ak ) @ r)
804      ==>
805      ( All id uk CDSK MDSK #i.
806          not( Ex #r. UpdateKeyReveal( uk ) @ r)
807          & not( Ex #r . CDSKReveal( CDSK ) @ r )
808          & not( Ex #r . MDSKReveal( MDSK ) @ r )
809          & Sourced_SKs( id, uk, CDSK, MDSK ) @ i
810           ==>  not( Ex #j . K( CDSK ) @ j ) & not( Ex #j. K( MDSK ) @ j ))"
811
812  lemma sessionkeys_sourced[reuse,use_induction]:
813      "not( Ex ak #r. AuthorityKeyReveal( ak ) @ r )
814      ==>
815      ( All linkid uk CDSK MDSK #i.
816          // If the Update key wasn't revealed, the session keys were set correctly
817          not( Ex #kr. UpdateKeyReveal( uk ) @ kr & #kr < #i )
818          & Sourced_SKs( linkid, uk, CDSK, MDSK ) @ i
819          ==> Ex #j MDSK2. #j < #i & NewSKs( linkid, uk, CDSK, MDSK2 ) @ j)"
820
821  lemma asdu_agreement_implies_mode_agreement[reuse]:
822      "not( Ex ak #r. AuthorityKeyReveal( ak ) @ r )
823      ==>
824      ( All linkid ar mode direction linkid2 mode2 direction2 #i #j.
825          ( All cdsk mdsk. UsingSessKeys( cdsk, mdsk ) @ i
826          ==> // The update key that was used to send out
827              // the current session keys cannot be revealed
828              ( All uk #k. UpdateKeyUsedForSKs( linkid, uk, cdsk, mdsk ) @ k
829                  ==> not( Ex #kr. UpdateKeyReveal( uk ) @ kr & #kr < #i ) )
830              // If the direction is control, then then no reveal of the current CDSK
831              & ( direction = 'control'
832                  ==> not( Ex #skr. CDSKReveal( cdsk ) @ skr & #skr < #i ) )
833              // And if the direction is monitor, then no reveal of the current MDSK
834              & ( direction = 'monitor'
835                  ==> not( Ex #skr. MDSKReveal( mdsk ) @ skr & #skr < #i ) ) )
836          & AuthASDU( linkid, ar, mode, direction ) @ i
837          & SentASDU( linkid2, ar, mode2, direction2 ) @ j & #j < #i
838          ==> ( mode = mode2 ) & ( direction = direction2 ) & ( linkid = linkid2 ))"
839
840  lemma asdu_aliveness[use_induction,hide_lemma=update_key_sourced]:
841      "not( Ex ak #r. AuthorityKeyReveal( ak ) @ r )
842      ==>
843      ( All linkid ar mode direction #i.
844          ( All cdsk mdsk. UsingSessKeys( cdsk, mdsk ) @ i
845          ==> // The update key that was used to send out
846              // the current session keys cannot be revealed
847              ( All uk #k. UpdateKeyUsedForSKs( linkid, uk, cdsk, mdsk ) @ k
848                  ==> not( Ex #kr. UpdateKeyReveal( uk ) @ kr & #kr < #i ) )
849              // If the direction is control, then then no reveal of the current CDSK
850              & ( direction = 'control'
851                  ==> not( Ex #skr. CDSKReveal( cdsk ) @ skr & #skr < #i ) )
852              // And if the direction is monitor, then no reveal of the current MDSK
853              & ( direction = 'monitor'
854                  ==> not( Ex #skr. MDSKReveal( mdsk ) @ skr & #skr < #i ) ) )
855          & AuthASDU( linkid, ar, mode, direction ) @ i
856              ==> Ex #j. SentASDU( linkid, ar, mode, direction ) @ j & j < i)"
857
858  lemma asdu_injective_agreement:
859      "not( Ex ak #r. AuthorityKeyReveal(ak) @ r )
860      ==>
861      ( All linkid ar mode direction #i #j.
862          ( All cdsk mdsk. UsingSessKeys( cdsk, mdsk ) @ i
863          ==> // The update key that was used to send out
864              // the current session keys cannot be revealed
865              ( All uk #k. UpdateKeyUsedForSKs( linkid, uk, cdsk, mdsk ) @ k
866                  ==> not( Ex #kr. UpdateKeyReveal( uk ) @ kr & #kr < #i ) )
867              // If the direction is control, then then no reveal of the current CDSK
868              & ( direction = 'control'
```

```
869                   ==> not( Ex #skr. CDSKReveal( cdsk ) @ skr & #skr < #i ) )
870               // And if the direction is monitor, then no reveal of the current MDSK
871               & ( direction = 'monitor'
872                   ==> not( Ex #skr. MDSKReveal( mdsk ) @ skr & #skr < #i ) ) )
873          & AuthASDU( linkid, ar, mode, direction ) @ i
874          & SentASDU( linkid, ar, mode, direction ) @ j & j < i
875      ==> not( Ex #k. AuthASDU( linkid, ar, mode, direction ) @ k & not( #k = #i ) ))"
876
877 end
```

A.3 Counter-based detection protocols

A.3.1 Example protocol

```
1  /*
2   * Protocol:    Counter-based detection example
3   * Modeler:     Kevin Milner
4   * Date:        May 2016
5   * Source:      Original
6   */
7
8  theory counter_example
9  begin
10
11 builtins: signing, hashing
12 functions: S/1
13
14 restriction Eq_testing: "All x y #i. Eq(x,y) @ i ==> x = y"
15 restriction InEq_testing: "All x y #i. InEq(x,y) @ i ==> not(x = y)"
16
17 //There's one id, per role, per direction.
18 restriction pairings_unique:
19     "All id1 id2 role ki kr #i #j.
20         Paired(id1,role,ki,kr) @ i & Paired(id2,role,ki,kr) @ j
21     ==> #i = #j"
22
23 rule RegisterKey:
24     [ Fr(~ltkA) ]
25   --[ Key(~ltkA) ]->
26     [ !F_AgentKey(~ltkA), Out(pk(~ltkA)) ]
27
28 rule CompromiseKey:
29     [ !F_AgentKey(~ltk) ]
30   --[ Compromise(pk(~ltk)) ]->
31     [ Out(~ltk) ]
32
33 /* Models an agent adding anothers public key out-of-band, we assume
34  * that all relationships set up this way are 'sane' in that both of the
35  * keys involved were generated fresh. */
36 rule BindState_Init:
37     [ !F_AgentKey(~ltkA)
38     , !F_AgentKey(~ltkB)
39     , Fr(~id)
40     ]--[
41       Paired(~id, 'I', ~ltkA, ~ltkB)
42     , Counter(~id, '0')
43     , F_InvariantSource_I(~id,~ltkA,pk(~ltkB))
44     ]->
45     [ /* For search efficiency, state is divided into
46        * an invariant portion and a variant portion. This
47        * allows tamarin to immediately bind the keys back
48        * to this initial pairing rule. The fresh ~id is used
49        * to identify this pairing. */
50       St_I(~id, ~ltkA, pk(~ltkB), 'm0', '0')
51     ]
52
53 rule BindState_Resp:
54     [ !F_AgentKey(~ltkA)
55     , !F_AgentKey(~ltkB)
56     , Fr(~id)
57     ]--[
58       Paired(~id, 'R', ~ltkA, ~ltkB)
59     , Counter(~id, '0')
60     , F_InvariantSource_R(~id,~ltkA,pk(~ltkB))
61     ]->
62     [ St_R(~id, ~ltkA, pk(~ltkB), 'm1', '0')
63     ]
```

```
64
65  ///////////////////////////////////////////
66  //   Message rules
67  rule I_m0:
68      [ St_I(~id, ~ltkI, pkR, 'm0', ic)
69      , Fr(~ni)
70      ]--[
71        Begin('I', <pk(~ltkI),pkR>, S(ic))
72      , Counter(<~id,'m0'>, ic)
73      , Gen(~ni)
74      , Injectivity(~id,ic,~ni)
75
76      , Invariant_I(~id,~ltkI,pkR)
77      ]->
78      [ St_I(~id, ~ltkI, pkR, <'m2',~ni>, ic)
79      , Out(~ni)
80      ]
81
82  rule R_m1:
83      let m1 = <'1', pk(~ltkR), pkI, ni, ~nr, S(rc)> in
84      [ St_R(~id, ~ltkR, pkI, 'm1', rc)
85      , Fr(~nr)
86      , In(ni)
87      ]--[
88        Begin('R', <pkI,pk(~ltkR)>, S(rc))
89      , Counter(<~id,'m1'>, rc)
90      , Nonces(~id,ni,~nr)
91      , Gen(~nr)
92      , Injectivity(~id,rc,~nr)
93
94      , Invariant_R(~id,~ltkR,pkI)
95      ]->
96      [ St_R(~id, ~ltkR, pkI, <'m2', ni, ~nr>, S(rc))
97      , Out(<m1, sign{m1}~ltkR>)
98      ]
99
100 rule I_m2:
101     let m1 = <'1', pkR, pk(~ltkI), ni, nr, src>
102         m2 = <'2', pk(~ltkI), pkR, ni, nr, src> in
103     [ St_I(~id, ~ltkI, pkR, <'m2', ni>, ic)
104     , In(<m1, sig>)
105     ]--[
106       Eq(verify(sig,m1,pkR), true)
107     , Eq(S(ic),src)
108     , Counter(~id,S(ic))
109     , Nonces(~id,ni,nr)
110     , Session('I', <pk(~ltkI),pkR>, <ni, nr>, src)
111     , Injectivity(~id,ni,S(ic))
112
113     , Invariant_I(~id,~ltkI,pkR)
114     ]->
115     [ St_I(~id, ~ltkI, pkR, 'm0', S(ic))
116     , Out(<m2, sign{m2}~ltkI>)
117     ]
118
119
120 rule I_detect:
121     let m1 = <'1', pkR, pk(~ltkI), ni, nr, src>
122         m2 = <'2', pk(~ltkI), pkR, ni, nr, src> in
123     [ St_I(~id, ~ltkI, pkR, <'m2', ni>, ic)
124     , In(<m1, sig>)
125     ]--[
126       Eq(verify(sig,m1,pkR), true)
127     , InEq(S(ic),src)
128     , Detect(<pk(~ltkI),pkR>)
129     , Counter(~id,S(ic))
130     , Nonces(~id,ni,nr)
131     , Session('I', <pk(~ltkI),pkR>, <ni, nr>, src)
132     , Injectivity(~id,ni,S(ic))
133
134     , Invariant_I(~id,~ltkI,pkR)
135     ]->
136     [ Remediate_Compromise(~id, ~ltkI, pkR, 'm0', S(ic))
137     , Discard(<m2, sign{m2}~ltkI>)
138     ]
139
140 rule R_m2:
141     let m2 = <'2', pkI, pk(~ltkR), ni, nr, src> in
142     [ St_R(~id, ~ltkR, pkI, <'m2', ni, nr>, src)
143     , Fr(~inj)
144     , In(<m2, sig>)
145     ]--[
146       Eq(verify(sig,m2,pkI), true)
147     , Completed(~id)
148     , Session('R', <pkI,pk(~ltkR)>, <ni,nr>, src)
```

```
149      , Counter(~id,src)
150      , Injectivity(~id,nr,src)
151 
152      , Invariant_R(~id,~ltkR,pkI)
153      ]->
154      [ St_R(~id, ~ltkR, pkI, 'm1', src)
155      ]
156 
157 
158 
159 ////////////////////////////////////////////
160 //   Helper Lemmas:
161 lemma invariant_sources[sources]:
162     "(All id ka kb #i.
163         Invariant_I(id,ka,kb) @ i
164     ==> Ex #j. F_InvariantSource_I(id,ka,kb) @ j & #j < #i)
165     &(All id ka kb #i.
166         Invariant_R(id,ka,kb) @ i
167     ==> Ex #j. F_InvariantSource_R(id,ka,kb) @ j & #j < #i)"
168 
169 lemma count_unique[reuse, use_induction]:
170     "All id c #i #j.
171         Counter(id,c) @ i & Counter(id,c) @ j
172     ==> #i = #j"
173 
174 lemma nonces_unique[reuse, use_induction]:
175     "All id ni nr #i #j.
176         Nonces(id,ni,nr) @ i & Nonces(id,ni,nr) @ j
177     ==> #i = #j"
178 
179 lemma force_nonce_ordering[reuse]:
180     "(All role pks ni nr c #i #j.
181         Session(role, pks, <ni,nr>, c) @ i & Gen(ni) @ j
182     ==> #j < #i)
183     &(All role pks ni nr c #i #j.
184         Session(role, pks, <ni,nr>, c) @ i & Gen(nr) @ j
185     ==> #j < #i)
186     &(All id ni nr #i #j.
187         Nonces(id, ni, nr) @ i & Gen(ni) @ j
188     ==> #j < #i)"
189 
190 
191 ////////////////////////////////////////////
192 //   Trace existence
193 lemma exists_session: exists-trace
194     "Ex pks nonces c #t0 #t1. #t0 < #t1 &
195         Session('I', pks, nonces, c) @ t0
196         & Session('R', pks, nonces, c) @ t1
197         & not(Ex #k key. Compromise(key) @ k) & not(Ex #k. Detect(pks) @ k)"
198 
199 lemma exists_second_session: exists-trace
200     "Ex pks nonces nonces2 c #t0 #t1 #t2 #t3. #t0 < #t2 & #t1 < #t3 & c = S('0') &
201         Session('I',pks,nonces,c) @ t0
202         & Session('R',pks,nonces,c) @ t1
203       & Session('I',pks,nonces2,S(c)) @ t2
204         & Session('R',pks,nonces2,S(c)) @ t3
205         & not(Ex #k key. Compromise(key) @ k) & not(Ex #k. Detect(pks) @ k)"
206 
207 lemma exists_detect_no_R_compromise: exists-trace
208     "Ex ki kr  #i.
209         Detect(<ki,kr>) @ i & not(Ex #j. Compromise(kr) @ j)"
210 
211 
212 lemma exists_detect_no_I_compromise: exists-trace
213     "Ex ki kr  #i.
214         Detect(<ki,kr>) @ i & not(Ex #j. Compromise(ki) @ j)"
215 
216 ////////////////////////////////////////////
217 //   Additional helper lemma for injectivity
218 
219 lemma injectivity[reuse]:
220     "All id n1 n2 n3 #i #j on1 on2 #k.
221         //Injective facts that are linked by n2
222         Injectivity(id,n1,n2) @ i & Injectivity(id,n2,n3) @ j
223         //And some other injective fact with the same id with relation to j
224         & Injectivity(id,on1,on2) @ k & #k < #j
225     ==> //can't be in between (might be equal to #i, though)
226         not(#i < #k)"
227 
228 ////////////////////////////////////////////
229 //  Trace properties
230 lemma detect_sound:
231     "All ki kr #t1.
232         Detect(<ki, kr>) @ t1
233     ==> (Ex #t0. #t0 < #t1 & Compromise(ki) @ t0)
234         | (Ex #t0. #t0 < #t1 & Compromise(kr) @ t0)"
```

```
235
236 //Protocol Property without compromise
237 lemma correct_dolevyao:
238     "All pks nonces c #t1.
239         Session('R',pks,nonces,c) @ t1 & not(Ex k #tc. Compromise(k) @ tc)
240     ==> Ex #t0. #t0 < #t1 &
241             Session('I',pks,nonces,c) @ t0"
242
243 lemma unmatching_implies_detect_with_R_uncompromised[use_induction]:
244     "All keys nonces1 nonces2 c1 c2 #t1 #t2 #t3.
245         #t1 < #t2 &
246         //Given an matching session
247         Begin('R', keys, c2) @ t2
248         & Session('I', keys, nonces2, c2) @ t3
249         // when I's key is not compromised
250         & not(Ex i r #tc. (keys = <i,r>) & Compromise(r) @ tc)
251         //Then for all sessions before that
252         & Session('R', keys, nonces1, c1) @ t1
253     ==> //Either that session was also matching
254         (Ex #t0. Session('I', keys, nonces1, c1) @ t0)
255         // Or I detected it was not
256         | (Detect(keys) @ t3)"
257
258 end
```

A.3.2 Modified Keyless SSL

```
1  /*
2   * Protocol:    Modified Keyless SSL protocol for Client Detection
3   * Modeler:   Kevin Milner
4   * Source:      Original
5   */
6
7  theory Keyless_SSL_Modified
8  begin
9
10 builtins: signing, hashing
11 functions: S/1
12
13 restriction Eq_testing: "All x y #i. Eq(x,y) @ i ==> x = y"
14 restriction InEq_testing: "All x y #i. InEq(x,y) @ i ==> not(x = y)"
15
16 //There's one id, per role, per direction.
17 restriction pairings_unique:
18     "All id1 id2 role ka kb #i #j.
19         Paired(id1,role,ka,kb) @ i & Paired(id2,role,ka,kb) @ j
20     ==> #i = #j"
21
22 rule RegisterKey:
23     [ Fr(~ltk) ]
24     --[Key(~ltk)]->
25     [!F_AgentKey(~ltk), Out(pk(~ltk))]
26
27 rule CompromiseKey:
28     [!F_AgentKey(~ltk)]
29     --[ Compromise(pk(~ltk)) ]->
30     [Out(~ltk)]
31
32
33 /* Models an agent adding anothers public key out-of-band, we assume
34  * that all relationships set up this way are 'sane' and both of the
35  * keys involved were generated fresh. */
36 rule BindState_C:
37     [ !F_AgentKey(~ltkA)
38     , !F_AgentKey(~ltkB)
39     , Fr(~id)
40     ]--[
41       Paired(~id, 'C', ~ltkA, ~ltkB)
42     , Counter(~id, '0')
43     , Injectivity(~id,'nonce','0')
44
45     , F_InvariantSource_C(~id,~ltkA,pk(~ltkB))
46     ]->
47     [ /* For search efficiency, state is divided into
48        * an invariant portion and a variant portion. This
49        * allows tamarin to immediately bind the keys back
50        * to this initial pairing rule. The fresh ~id is used
51        * to identify this pairing. */
52       St_C(~id, ~ltkA, pk(~ltkB), 'm0', '0')
53     ]
54
```

```
55  rule BindState_W:
56      [ !F_AgentKey(~ltkA)
57      , !F_AgentKey(~ltkB)
58      , Fr(~id)
59      ]--[
60        Paired(~id, 'W', ~ltkA, ~ltkB)
61      , Counter(~id, '0')
62      , Injectivity(~id,'nonce','0')
63
64      , F_InvariantSource_W(~id,~ltkB,pk(~ltkA))
65      ]->
66      [ St_W(~id, ~ltkB, pk(~ltkA), 'm1', '0')
67      ]
68
69  /////////////////////////////////////////
70  // MESSAGE RULES
71  rule C_0:
72      let m1 = <'1',pk(~ltkA), pkB,~nc> in
73      [ St_C(~id, ~ltkA, pkB, 'm0', cc)
74      , Fr(~nc)
75      ]--[
76        Gen(~nc)
77      , Injectivity(~id,cc,~nc)
78
79      , Invariant_C(~id,~ltkA,pkB)
80      ]->
81      [ St_C(~id, ~ltkA, pkB, <'m2', ~nc>, cc)
82      , Out(m1)
83      ]
84
85  rule W_0:
86      let pkB = pk(~ltkB)
87          m1 = <'1',pkA,pkB,nc>
88          m2 = <'2',pkB,pkA,nc,~nw> in
89      [ St_W(~id, ~ltkB, pkA, 'm1', cw)
90      , Fr(~nw)
91      , In(m1)
92      ]--[
93        Nonces(~id,nc,~nw)
94      , Counter(<~id,'m1'>, cw)
95      , Gen(~nw)
96      , Begin('W', <pkA,pkB>, S(cw))
97      , Injectivity(~id,cw,~nw)
98
99      , Invariant_W(~id,~ltkB,pkA)
100     ]->
101     [ St_W(~id, ~ltkB, pkA, <'m3', nc, ~nw>, cw)
102     , Out(<m2,sign{m2}~ltkB>)
103     ]
104
105 rule C_1:
106     let pkA = pk(~ltkA)
107         m1 = <'1',pkA,pkB,nc>
108         m2 = <'2',pkB,pkA,nc,nw>
109         m3 = <'3',pkA,pkB,h(<m1,m2>),S(cc)> in
110     [ St_C(~id, ~ltkA, pkB, <'m2', nc>, cc)
111     , In(<m2,sig>)
112     , Fr(~inj) // Just for maintaining injectivity actions
113     ]--[
114       Eq(verify(sig,m2,pkB), true)
115     , Nonces(~id,nc,nw)
116     , Counter(<~id,'m3'>, S(cc))
117     , Injectivity(~id,nc,~inj)
118     , Begin('C', <pkA,pkB>, S(cc))
119
120     , Invariant_C(~id,~ltkA,pkB)
121     ]->
122     [ St_C(~id, ~ltkA, pkB, <'m4', nc, nw, ~inj>, S(cc))
123     , Out(<m3, sign{m3}~ltkA>)
124     ]
125
126
127
128 rule W_detect:
129     let pkB = pk(~ltkB)
130         m1 = <'1',pkA,pkB,nc>
131         m2 = <'2',pkB,pkA,nc,nw>
132         m3 = <'3',pkA,pkB,h(<m1,m2>),scc>
133         m4 = <'4',pkB,pkA,h(<m1,m2,m3>)> in
134     [ St_W(~id, ~ltkB, pkA, <'m3', nc, nw>, cw)
135     , In(<m3, sig>)
136     ]--[
137       Eq(verify(sig,m3,pkA), true)
138     , InEq(scc, S(cw))
139     , Detect(<pkA,pkB>)
140     , Session('W', <pkA,pkB>, <nc,nw>, scc)
```

```
141      , Counter(<~id, 'm4'>, S(cw))
142      , Injectivity(~id,nw,S(cw))
143
144      , Invariant_W(~id,~ltkB,pkA)
145      ]->
146      [ Remediate_W(~id, ~ltkB, pkA, 'm1', scc)
147      , Discard(<m4, sign{m4}~ltkB>)
148      ]
149
150  rule W_session:
151      let pkB = pk(~ltkB)
152          m1 = <'1',pkA,pkB,nc>
153          m2 = <'2',pkB,pkA,nc,nw>
154          m3 = <'3',pkA,pkB,h(<m1,m2>),scc>
155          m4 = <'4',pkB,pkA,h(<m1,m2,m3>)> in
156      [ St_W(~id, ~ltkB, pkA, <'m3', nc, nw>, cw)
157      , In(<m3, sig>)
158      ]--[
159        Eq(verify(sig,m3,pkA), true)
160      , Eq(scc, S(cw))
161      , Completed(~id)
162      , Session('W', <pkA,pkB>, <nc,nw>, scc)
163      , Counter(<~id, 'm4'>, S(cw))
164      , Injectivity(~id,nw,S(cw))
165
166      , Invariant_W(~id,~ltkB,pkA)
167      ]->
168      [ St_W(~id, ~ltkB, pkA, 'm1', scc)
169      , Out(<m4, sign{m4}~ltkB>)
170      ]
171
172
173  rule C_2:
174      let pkA = pk(~ltkA)
175          m1 = <'1',pkA,pkB,nc>
176          m2 = <'2',pkB,pkA,nc,nw>
177          m3 = <'3',pkA,pkB,h(<m1,m2>),scc>
178          m4 = <'4',pkB,pkA,h(<m1,m2,m3>)> in
179      [ St_C(~id, ~ltkA, pkB, <'m4', nc, nw, ~inj>, scc)
180      , In(<m4, sig>)
181      , Fr(~injc)
182      ]--[
183        Eq(verify(sig,m4,pkB), true)
184      , Counter(<~id,'m4'>, scc)
185      , Complete('C', <pkA,pkB>, <nc,nw>, scc)
186      , Injectivity(~id,~inj,scc)
187      , Session('C', <pkA,pkB>, <nc,nw>, scc)
188
189      , Invariant_C(~id,~ltkA,pkB)
190      ]->
191      [ St_C(~id, ~ltkA, pkB, 'm0', scc)
192      ]
193
194
195  //////////////////////////////////////////
196  //   Helper Lemmas:
197  lemma invariant_sources[sources]:
198      "(All id ka kb #i.
199          Invariant_C(id,ka,kb) @ i
200      ==> Ex #j. F_InvariantSource_C(id,ka,kb) @ j & #j < #i)
201      &(All id ka kb #i.
202          Invariant_W(id,ka,kb) @ i
203      ==> Ex #j. F_InvariantSource_W(id,ka,kb) @ j & #j < #i)"
204
205  lemma count_unique[reuse, use_induction]:
206      "All id c #i #j.
207          Counter(id,c) @ i & Counter(id,c) @ j
208      ==> #i = #j"
209
210  lemma nonces_unique[reuse, use_induction]:
211      "All id ni nr #i #j.
212          Nonces(id,ni,nr) @ i & Nonces(id,ni,nr) @ j
213      ==> #i = #j"
214
215  lemma force_nonce_ordering[reuse]:
216      "(All role pks nc nw c #i #j.
217          Session(role, pks, <nc,nw>, c) @ i & Gen(nc) @ j
218      ==> #j < #i)
219      &(All role pks nc nw c #i #j.
220          Session(role, pks, <nc,nw>, c) @ i & Gen(nw) @ j
221      ==> #j < #i)
222      &(All id nc nw #i #j.
223          Nonces(id, nc, nw) @ i & Gen(nc) @ j
224      ==> #j < #i)"
225
226  //////////////////////////////////////////
```

```
227 //   Trace existence
228 lemma exists_session: exists-trace
229     "Ex pks nonces c #t0 #t1. #t0 < #t1 &
230         Session('W', pks, nonces, c) @ t0
231         & Complete('C', pks, nonces, c) @ t1
232         & not(Ex #k key. Compromise(key) @ k) & not(Ex #k. Detect(pks) @ k)"
233
234 lemma exists_second_session: exists-trace
235     "Ex pks nonces nonces2 c #t0 #t1 #t2 #t3. #t0 < #t2 & #t1 < #t3 & c = S('0') &
236         Session('W',pks,nonces,c) @ t0
237         & Complete('C',pks,nonces,c) @ t1
238       & Session('W',pks,nonces2,S(c)) @ t2
239         & Complete('C',pks,nonces2,S(c)) @ t3
240         & not(Ex #k key. Compromise(key) @ k) & not(Ex #k. Detect(pks) @ k)"
241
242 lemma exists_detect_no_C_compromise: exists-trace
243     "Ex kc kw  #i.
244         Detect(<kc,kw>) @ i & not(Ex #j. Compromise(kc) @ j)"
245
246 lemma exists_detect_no_I_compromise: exists-trace
247     "Ex kc kw  #i.
248         Detect(<kc,kw>) @ i & not(Ex #j. Compromise(kw) @ j)"
249
250 /////////////////////////////////////////
251 //   Additional helper lemma for injectivity
252 lemma injectivity[reuse]:
253     "All id n1 n2 n3 #i #j on1 on2 #k.
254         //Injective facts that are linked by n2
255         Injectivity(id,n1,n2) @ i & Injectivity(id,n2,n3) @ j
256         //And some other injective fact with the same id with relation to j
257         & Injectivity(id,on1,on2) @ k & #k < #j
258     ==> //can't be in between (might be equal to #i, though)
259         not(#i < #k)"
260
261 /////////////////////////////////////////
262 //  Trace properties
263 lemma detect_sound:
264     "All kc kw #t1.
265         Detect(<kc, kw>) @ t1
266     ==> (Ex #t0. #t0 < #t1 & Compromise(kc) @ t0)
267         | (Ex #t0. #t0 < #t1 & Compromise(kw) @ t0)"
268
269 lemma correct_dolevyao:
270     "All pks nonces c #t1.
271         Complete('C',pks,nonces,c) @ t1 & not(Ex k #tc. Compromise(k) @ tc)
272     ==> Ex #t0. #t0 < #t1 &
273             Session('W',pks,nonces,c) @ t0"
274
275 lemma unmatching_implies_detect_with_W_uncompromised[use_induction]:
276     "All keys nonces1 nonces2 c1 c2 #t1 #t2 #t3.
277         #t0 < #t3 &
278         //Given an matching m3
279         Begin('C', keys, c2) @ t2
280         & Session('W', keys, nonces2, c2) @ t3
281         // And the web services key is uncompromised
282         & not(Ex kc kw #tc. (keys = <kc,kw>) & Compromise(kw) @ tc)
283         //Then for all sessions before that
284         & Session('W', keys, nonces1, c1) @ t0
285     ==> //Either there was a corresponding C request
286         (Ex #t0. Session('C', keys, nonces1, c1) @ t1)
287         //Or W detects was compromised
288         | (Detect(keys) @ t3)"
289
290 end
```

A.4 Commitment-based detection protocols

A.4.1 Example protocol

```
1
2
3 /*
4  * Protocol:    Token commitment protocol
5  * Modeler:     Kevin Milner
6  * Date:        May 2016
7  * Source:      Original
8  */
9
```

```
10  theory commitment_example
11  begin
12
13  builtins: signing
14
15  restriction Eq_testing: "All x y #i. Eq(x,y) @ i ==> x = y"
16  restriction InEq_testing: "All x y #i. InEq(x,y) @ i ==> not(x = y)"
17
18  //Limit to one id, per role, per direction.
19  restriction pairings_unique:
20      "All id1 id2 role ki kr #i #j.
21          Paired(id1,role,ki,kr) @ i & Paired(id2,role,ki,kr) @ j
22      ==> #i = #j"
23
24  rule RegisterKey:
25      [ Fr(~ltkA) ]
26    --[ Key(~ltkA) ]->
27      [ !F_AgentKey(~ltkA), Out(pk(~ltkA)) ]
28
29  rule CompromiseKey:
30      [ !F_AgentKey(~ltk) ]
31    --[ Compromise(pk(~ltk)) ]->
32      [ Out(~ltk) ]
33
34  rule State_Compromise:
35      [ !F_TokenState(pkA,pkB,curtok)
36      ]--[
37        CompromiseToken(<pkA,pkB>,curtok)
38      ]->
39      [ Out(curtok)
40      ]
41
42  /* Models an agent adding anothers public key out-of-band, we assume
43   * that all relationships set up this way are 'sane' in that both of the
44   * keys involved were generated fresh. */
45  rule BindState_Init:
46      [ !F_AgentKey(~ltkA)
47      , !F_AgentKey(~ltkB)
48      , Fr(~id)
49      ]--[
50        Paired(~id, 'I', ~ltkA, ~ltkB)
51      , Counter(~id, '0')
52      , InvariantSource_I(~id,~ltkA,pk(~ltkB))
53      ]->
54      [
55        St_I(~id, ~ltkA, pk(~ltkB), 'm1',
56          'uninitialized', sign{'uninitialized'}'notoken')
57      ]
58
59  rule BindState_Resp:
60      [ !F_AgentKey(~ltkA)
61      , !F_AgentKey(~ltkB)
62      , Fr(~id)
63      ]--[
64        Paired(~id, 'R', ~ltkA, ~ltkB)
65      , Counter(~id, '0')
66      , InvariantSource_R(~id,~ltkB,pk(~ltkA))
67      ]->
68      [ St_R(~id,~ltkB,pk(~ltkA), sign{'uninitialized'}'notoken', 'notoken')
69      ]
70
71  /////////////////////////////////////////
72  //   Message rules
73  rule I_1:
74      let pkA = pk(~ltkA)
75          m1 = <'1',pkA, pkB,~ni,commit> in
76      [ St_I(~id, ~ltkA, pkB,'m1',commitmsg,commit)
77      , Fr(~ni)
78      ]--[
79        Begin('I',<pkA,pkB>,~ni,commit)
80      , Gen(~ni)
81      , Invariant_I(~id,~ltkA,pkB)
82      ]->
83      [ St_I(~id, ~ltkA, pkB,<'m2',~ni>,commitmsg,commit)
84      , Out(<m1, sign{m1}~ltkA>)
85      ]
86
87  rule R_2:
88      let pkB = pk(~ltkB)
89          m1 = <'1',pkA,pkB,ni,icommit>
90          //Sign nonce to prove we still have secret key
91          //Provide our proof of the previous commit with the public key
92          //Provide a new commit dependent on some known freshness from I
93          newcommit = sign{<'commit',pkB,pkA,ni>}~newtok
94          ncc = <ni,icommit,newcommit>
95          m2 = <'2',pkB,pkA,pk(token),newcommit,sign{ncc}token> in
```

```
96      [ St_R(~id,~ltkB,pkA,commit,token)
97      , Fr(~newtok)
98      , In(<m1, sig>)
99      ]--[
100       Eq(verify(sig,m1,pkA), true)
101     , Token(~newtok)
102     , Commit(<pkA,pkB>,newcommit)
103     , Eq(icommit, commit)         //Matches the commit we expect in I's request
104     , Session('R', <pkA,pkB>, <ni,icommit,pk(token),newcommit>)
105
106     , Invariant_R(~id,~ltkB,pkA)
107     ]->
108     [ St_R(~id,~ltkB,pkA,newcommit,~newtok)
109     , !F_TokenState(pkA,pkB,~newtok)
110     , Out(<m2,sign{m2}~ltkB>)
111     ]
112
113
114
115 //Everything matches as expected
116 rule I_completed:
117     let pkA = pk(~ltkA)
118         m2 = <'2',pkB,pkA,proof,newcommit,nccsig> in
119     [ St_I(~id, ~ltkA, pkB,<'m2',ni>,commitmsg,commit)
120     , In(<m2,sig>)
121     ]--[
122       Eq(verify(sig,m2,pkB), true)
123     , InEq(newcommit,commit)      //Needs to be a new signature
124     , Session('I', <pkA,pkB>, <ni,commit,proof,newcommit>)
125     //Check token matches previous commit:
126     , Eq(verify(commit,commitmsg,proof),true)
127     //Check the nonce signature to ensure R is alive:
128     , Eq(verify(nccsig,<ni,commit,newcommit>,proof),true)
129     //Session or detect depending on the previous checks:
130     , Complete(<pkA,pkB>)
131
132     , Invariant_I(~id,~ltkA,pkB)
133     ]->
134     [ St_I(~id, ~ltkA, pkB,'m1', <'commit',pkB,pkA,ni>, newcommit)
135     ]
136
137
138 //Proof mismatch but signed with B's key, so it must have been compromised.
139 rule I_detect:
140     let pkA = pk(~ltkA)
141         m2 = <'2',pkB,pkA,proof,newcommit,nccsig> in
142     [ St_I(~id, ~ltkA, pkB,<'m2',ni>,commitmsg,commit)
143     , In(<m2,sig>)
144     ]--[
145       Eq(verify(sig,m2,pkB), true)
146     , InEq(newcommit,commit)      //Needs to be a new signature
147     , Session('I', <pkA,pkB>, <ni,commit,proof,newcommit>)
148     //Check token matches previous commit:
149     , InEq(verify(commit,commitmsg,proof),true)
150     //Check the nonce signature to ensure R is alive:
151     , Eq(verify(nccsig,<ni,commit,newcommit>,proof),true)
152     //Session or detect depending on the previous checks:
153     , Detect(<pkA,pkB>)
154
155     , Invariant_I(~id,~ltkA,pkB)
156     ]->
157     [ Remediate_I(~id, ~ltkA, pkB,'m1', <'commit',pkB,pkA,ni>, newcommit)
158     ]
159
160
161 ///////////////////////////////////////////
162 //   Helper Lemmas:
163 lemma invariant_sources[sources]:
164     "(All id ka kb #i.
165         Invariant_I(id,ka,kb) @ i
166     ==> Ex #j. InvariantSource_I(id,ka,kb) @ j & #j < #i)
167     &(All id ka kb #i.
168         Invariant_R(id,ka,kb) @ i
169     ==> Ex #j. InvariantSource_R(id,ka,kb) @ j & #j < #i)"
170
171 lemma commit_unique[reuse]:
172     "All id1 id2 com #i #j.
173         Commit(id1,com) @ i & Commit(id2,com) @ j
174     ==> #i = #j & id1 = id2"
175
176 lemma token_compromise_source[reuse]:
177     "All pks tok #i.
178         CompromiseToken(pks,tok) @ i
179     ==> Ex #j. #j < #i & Token(tok) @ j"
180
181 lemma force_nonce_ordering[reuse]:
```

```
182      "(All role keys ni data #i #j.
183          Session(role, keys, <ni, data>) @ i & Gen(ni) @ j
184      ==> #j < #i)"
185
186  //////////////////////////////////////////
187  //   Trace existence
188  lemma exists_session: exists-trace
189      "Ex pks data #t0 #t1. #t0 < #t1 &
190          Session('R', pks, data) @ t0
191          & Session('I', pks, data) @ t1
192          & not(Ex #k key. Compromise(key) @ k) & not(Ex #k. Detect(pks) @ k)"
193
194  lemma exists_second_session: exists-trace
195      "Ex pks data data2 #t0 #t1 #t2 #t3. #t0 < #t2 & #t1 < #t3 &
196          Session('R',pks,data) @ t0
197          & Session('I',pks,data) @ t1
198        & Session('R',pks,data2) @ t2
199          & Session('I',pks,data2) @ t3
200          & not(Ex #k key. Compromise(key) @ k) & not(Ex #k. Detect(pks) @ k)"
201
202  lemma exists_detect_no_I_compromise: exists-trace
203      "Ex ki kr  #i.
204          Detect(<ki,kr>) @ i & not(Ex #j. Compromise(ki) @ j)"
205
206  //////////////////////////////////////////
207  //  Trace properties
208  lemma detect_sound:
209      "All ki kr #t1.
210          Detect(<ki, kr>) @ t1
211      ==> Ex #t0. #t0 < #t1 & Compromise(kr) @ t0"
212
213  lemma correct_dolevyao:
214      "All pks data #t1.
215          Session('I',pks,data) @ t1 & not(Ex k #tc. Compromise(k) @ tc)
216      ==> Ex #t0. #t0 < #t1 &
217              Session('R',pks,data) @ t0"
218
219  lemma sent_commit_implies_generated[reuse]:
220      "All pks pks2 n c data #t0 #t1.
221          Session('R',pks,<n,c,data>) @ t1
222          & Commit(pks2,c) @ t0
223      ==> (pks = pks2) & #t0 < #t1"
224
225  lemma sessions_injective[reuse, use_induction]:
226      "(All role pks n2 c2 data data2 #t0 #t1 #t2. #t0 < #t2 &
227          Session(role, pks, data2) @ t0
228          & Session(role, pks, <n2, c2, data>) @ t2
229          & Begin(role, pks, n2, c2) @ t1
230      ==> #t0 < #t1)
231      &(All role pks n c data n2 data2 #t0 #t1 #t2. #t0 < #t2 &
232          Begin(role, pks, n, c) @ t0
233          & Session(role, pks, <n, c, data>) @ t1
234          & Session(role, pks, <n2,data2>) @ t2
235          & not(n = n2)
236      ==> #t1 < #t2)"
237
238  lemma matching_detects_prior_misuse[use_induction]:
239      "All pks data data2 #t1 #t2 #t3.
240          #t1 < #t2 & #t2 < #t3 &
241          //Given an matching session
242          Session('R', pks, data2) @ t2 & Session('I', pks, data2) @ t3
243          //Where at least one key was uncompromised
244          & not(Ex i r #tc #tc2. (pks = <i,r>)
245              & Compromise(i) @ tc & Compromise(r) @ tc2)
246          //Then for all sessions before that
247          & Session('I', pks, data) @ t1
248      ==> //Either that session was also matching
249          (Ex #t0. #t0 < #t1 &
250              Session('R', pks, data) @ t0)
251          //Or I will detect
252          | (Detect(pks) @ t3)"
253
254  lemma matching_detects_later_misuse[use_induction]:
255      "All pks data data2 #t0 #t1 #t3.
256          #t0 < #t1 & #t1 < #t3
257          //Given a matching session
258          & Session('R',pks,data) @ t0
259          & Session('I',pks,data) @ t1
260          //and I finished a session without detecting
261          & Session('I',pks,data2) @ t3
262          & not(Detect(pks) @ t3)
263          //Without state compromise in between
264          & not(Ex #tc tok3. #tc < #t3 & #t0 < #tc &
265                  CompromiseToken(pks,tok3) @ tc)
266      ==> //Then there is an agreeing R session, regardless of key compromise
267          (Ex #t2. #t2 < #t3 &
```

```
268                 Session('R',pks,data2) @ t2)"
269
270 end
```

A.4.2 Modified ISO-IEC 9798-3-3 protocol

```
1  /*
2     Protocol:    Token commitment protocol applied to ISO-IEC-9798-3-3
3     Modeler:     Kevin Milner
4     Date:        July 2016
5     Source:      Original
6   */
7
8  theory ISO_IEC9798_3_3_Modified
9  begin
10
11 builtins: signing
12
13 restriction Eq_testing: "All x y #i. Eq(x,y) @ i ==> x = y"
14 restriction InEq_testing: "All x y #i. InEq(x,y) @ i ==> not(x = y)"
15
16 //There's one id, per role, per direction.
17 restriction pairings_unique:
18     "All id1 id2 role ki kr #i #j.
19         Paired(id1,role,ki,kr) @ i & Paired(id2,role,ki,kr) @ j
20     ==> #i = #j"
21
22 rule RegisterKey:
23     [ Fr(~ltkA) ]
24   --[ Key(~ltkA) ]->
25     [ !F_AgentKey(~ltkA), Out(pk(~ltkA)) ]
26
27 rule CompromiseKey:
28     [ !F_AgentKey(~ltk) ]
29   --[ Compromise(pk(~ltk)) ]->
30     [ Out(~ltk) ]
31
32 rule State_Compromise:
33     [ !F_TokenState(pkA,pkB,curtok)
34     ]--[
35       CompromiseToken(<pkA,pkB>,curtok)
36     ]->
37     [ Out(curtok)
38     ]
39
40 /* Models an agent adding anothers public key out-of-band, we assume
41  * that all relationships set up this way are 'sane' in that both of the
42  * keys involved were generated fresh. */
43 rule BindState_Init:
44     [ !F_AgentKey(~ltkA)
45     , !F_AgentKey(~ltkB)
46     , Fr(~id)
47     ]--[
48       Paired(~id, 'I', ~ltkA, ~ltkB)
49     , Counter(~id, '0')
50
51     , InvariantSource_I(~id,~ltkA,pk(~ltkB))
52     ]->
53     [ /* For search efficiency, state is divided into
54        * an invariant portion and a variant portion. This
55        * allows tamarin to immediately bind the keys back
56        * to this initial pairing rule. The fresh ~id is used
57        * to identify this pairing. */
58       St_I(~id,~ltkA,pk(~ltkB),'m1','uninitialized',sign{'uninitialized'}'notoken')
59     ]
60
61 rule BindState_Resp:
62     [ !F_AgentKey(~ltkA)
63     , !F_AgentKey(~ltkB)
64     , Fr(~id)
65     ]--[
66       Paired(~id, 'R', ~ltkA, ~ltkB)
67     , Counter(~id, '0')
68     , InvariantSource_R(~id,~ltkB,pk(~ltkA))
69     ]->
70     [ St_R(~id, ~ltkB, pk(~ltkA), sign{'uninitialized'}'notoken', 'notoken')
71     ]
72
73 ////////////////////////////////////////////
74 //    Message rules
75 rule I_1:
```

```
76      let pkA = pk(~ltkA)
77          m1 = <'1',pkA, pkB,~ni,commit> in
78      [ St_I(~id, ~ltkA, pkB,'m1',commitmsg,commit)
79      , Fr(~ni)
80      ]--[
81        Begin('I',<pkA,pkB>,~ni,commit)
82      , Gen(~ni)
83
84      , Invariant_I(~id,~ltkA,pkB)
85      ]->
86      [ St_I(~id, ~ltkA, pkB,<'m2',~ni>,commitmsg,commit)
87      , Out(<m1, sign{m1}~ltkA>)
88      ]
89
90
91  rule R:
92      let pkB = pk(~ltkB)
93          m1 = <'1',pkA,pkB,ni,icommit>
94          //Sign nonce to prove we still have secret key
95          //Provide our proof of the previous commit with the public key
96          //Provide a new commit dependent on some known freshness from I
97          newcommit = sign{<'commit',pkB,pkA,ni,~nr>}~newtok
98          ncc = <ni,icommit,~nr,newcommit>
99          m2 = <'2',pkB,pkA,~nr,ni,pk(token),newcommit,sign{ncc}token> in
100     [ St_R(~id, ~ltkB, pkA, commit,token)
101     , Fr(~newtok), Fr(~nr)
102     , In(<m1, sig>)
103     ]--[
104       Eq(verify(sig,m1,pkA), true)
105     , Gen(~nr)
106     , Token(~newtok)
107     , Commit(<pkA,pkB>,newcommit)
108     , Eq(icommit, commit)        //Matches the commit we expect in I's request
109     , Session('R', <pkA,pkB>, <ni,icommit,~nr,pk(token),newcommit>)
110
111     , Invariant_R(~id,~ltkB,pkA)
112     ]->
113     [ St_R(~id, ~ltkB, pkA, newcommit,~newtok)
114     , !F_TokenState(pkA,pkB,~newtok)
115     , Out(<m2,sign{m2}~ltkB>)
116     ]
117
118
119
120 //Everything matches as expected
121 rule I_2_complete:
122     let pkA = pk(~ltkA)
123         m2 = <'2',pkB,pkA,nr,ni,proof,newcommit,nccsig> in
124     [ St_I(~id, ~ltkA, pkB, <'m2',ni>,commitmsg,commit)
125     , In(<m2,sig>)
126     ]--[
127       Eq(verify(sig,m2,pkB), true)
128     , InEq(newcommit,commit)
129     , Session('I', <pkA,pkB>, <ni,commit,nr,proof,newcommit>)
130     //Check token matches previous commit:
131     , Eq(verify(commit,commitmsg,proof),true)
132     //Check the nonce signature to ensure R is alive:
133     , Eq(verify(nccsig,<ni,commit,nr,newcommit>,proof),true)
134     //Session or detect depending on the previous checks:
135     , Complete(<pkA,pkB>)
136
137     , Invariant_I(~id,~ltkA,pkB)
138     ]->
139     [ St_I(~id, ~ltkA, pkB, 'm1', <'commit',pkB,pkA,ni,nr>, newcommit)
140     ]
141
142
143 //Proof mismatch but signed with B's key, so it must have been compromised.
144 rule I_2_detect:
145     let pkA = pk(~ltkA)
146         m2 = <'2',pkB,pkA,nr,ni,proof,newcommit,nccsig> in
147     [ St_I(~id, ~ltkA, pkB, <'m2',ni>,commitmsg,commit)
148     , In(<m2,sig>)
149     ]--[
150       Eq(verify(sig,m2,pkB), true)
151     , InEq(newcommit,commit)      //Needs to be a new signature
152     , Session('I', <pkA,pkB>, <ni,commit,nr,proof,newcommit>)
153     //Check token matches previous commit:
154     , InEq(verify(commit,commitmsg,proof),true)
155     //Check the nonce signature to ensure R is alive:
156     , Eq(verify(nccsig,<ni,commit,nr,newcommit>,proof),true)
157     //Session or detect depending on the previous checks:
158     , Detect(<pkA,pkB>)
159
160     , Invariants_I(~id,~ltkA,pkB)
161     ]->
```

```
162      [ Remediate_R(~id, ~ltkA, pkB,'m1', <'commit',pkB,pkA,ni>, newcommit)
163      ]
164
165
166
167  //////////////////////////////////////////
168  //   Helper Lemmas:
169  lemma invariant_sources[sources]:
170      "(All id ka kb #i.
171          Invariant_I(id,ka,kb) @ i
172      ==> Ex #j. InvariantSource_I(id,ka,kb) @ j & #j < #i)
173      &(All id ka kb #i.
174          Invariant_R(id,ka,kb) @ i
175      ==> Ex #j. InvariantSource_R(id,ka,kb) @ j & #j < #i)"
176
177  lemma commit_unique[reuse]:
178      "All id1 id2 com #i #j.
179          Commit(id1,com) @ i & Commit(id2,com) @ j
180      ==> #i = #j & id1 = id2"
181
182  lemma token_compromise_source[reuse]:
183      "All pks tok #i.
184          CompromiseToken(pks,tok) @ i
185      ==> Ex #j. #j < #i & Token(tok) @ j"
186
187  lemma force_nonce_ordering[reuse]:
188      "(All role keys ni data #i #j.
189          Session(role, keys, <ni, data>) @ i & Gen(ni) @ j
190      ==> #j < #i)
191      &(All keys ni c nr data #i #j.
192          Session('I', keys, <ni,c,nr,data>) @ i & Gen(nr) @ j
193      ==> #j < #i)"
194
195  //////////////////////////////////////////
196  //   Trace existence
197  lemma exists_session: exists-trace
198      "Ex pks data #t0 #t1. #t0 < #t1 &
199          Session('R', pks, data) @ t0
200          & Session('I', pks, data) @ t1
201          & not(Ex #k key. Compromise(key) @ k) & not(Ex #k. Detect(pks) @ k)"
202
203  lemma exists_second_session: exists-trace
204      "Ex pks data data2 #t0 #t1 #t2 #t3. #t0 < #t2 & #t1 < #t3 &
205          Session('R',pks,data) @ t0
206          & Session('I',pks,data) @ t1
207        & Session('R',pks,data2) @ t2
208          & Session('I',pks,data2) @ t3
209          & not(Ex #k key. Compromise(key) @ k) & not(Ex #k. Detect(pks) @ k)"
210
211  lemma exists_detect_no_I_compromise: exists-trace
212      "Ex ki kr  #i.
213          Detect(<ki,kr>) @ i & not(Ex #j. Compromise(ki) @ j)"
214
215  //////////////////////////////////////////
216  //  Trace properties
217  lemma detect_sound:
218      "All ki kr #t1.
219          Detect(<ki, kr>) @ t1
220      ==> Ex #t0. #t0 < #t1 & Compromise(kr) @ t0"
221
222  lemma correct_dolevyao:
223      "All pks data #t1.
224          Session('I',pks,data) @ t1 & not(Ex k #tc. Compromise(k) @ tc)
225      ==> Ex #t0. #t0 < #t1 &
226              Session('R',pks,data) @ t0"
227
228
229  lemma sent_commit_implies_generated[reuse]:
230      "All pks pks2 n c data #t0 #t1.
231          Session('R',pks,<n,c,data>) @ t1
232          & Commit(pks2,c) @ t0
233      ==> (pks = pks2) & #t0 < #t1"
234
235
236  lemma sessions_injective[reuse, use_induction]:
237      "(All role pks n2 c2 data data2 #t0 #t1 #t2. #t0 < #t2 &
238          Session(role, pks, data2) @ t0
239          & Session(role, pks, <n2, c2, data>) @ t2
240          & Begin(role, pks, n2, c2) @ t1
241      ==> #t0 < #t1)
242      &(All role pks n c data n2 data2 #t0 #t1 #t2. #t0 < #t2 &
243          Begin(role, pks, n, c) @ t0
244          & Session(role, pks, <n, c, data>) @ t1
245          & Session(role, pks, <n2,data2>) @ t2
246          & not(n = n2)
247      ==> #t1 < #t2)"
```

```
248
249 lemma matching_detects_prior_misuse[use_induction]:
250     "All pks data data2 #t1 #t2 #t3.
251         #t1 < #t2 & #t2 < #t3 &
252         //Given an matching session
253         Session('R', pks, data2) @ t2 & Session('I', pks, data2) @ t3
254         //Where at least one key was uncompromised
255         & not(Ex i r #tc #tc2. (pks = <i,r>)
256             & Compromise(i) @ tc & Compromise(r) @ tc2)
257         //Then for all sessions before that
258         & Session('I', pks, data) @ t1
259     ==> //Either that session was also matching
260         (Ex #t0. #t0 < #t1 &
261             Session('R', pks, data) @ t0)
262         //Or I will detect
263         | (Detect(pks) @ t3)"
264
265 lemma matching_detects_later_misuse[use_induction]:
266     "All pks data data2 #t0 #t1 #t3.
267         #t0 < #t1 & #t1 < #t3
268         //Given a matching session
269         & Session('R',pks,data) @ t0
270         & Session('I',pks,data) @ t1
271         //and I finished a session without detecting
272         & Session('I',pks,data2) @ t3
273         & not(Detect(pks) @ t3)
274         //Without state compromise in between
275         & not(Ex #tc tok3. #tc < #t3 & #t0 < #tc &
276                 CompromiseToken(pks,tok3) @ tc)
277     ==> //Then there is an agreeing R session, regardless of key compromise
278         (Ex #t2. #t2 < #t3 &
279             Session('R',pks,data2) @ t2)"
280
281 end
```