CHALMERS
UNIVERSITY OF TECHNOLOGY

A Software Architecture to Ensure Surveillance Accountability

Master's thesis in Software Engineering

MUKELABAI MUKELABAI

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Master's thesis 2016:NN

A Software Architecture to Ensure Surveillance Accountability

MUKELABAI MUKELABAI

Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
Gothenburg, Sweden 2016

A Software Architecture to Ensure Surveillance Accountability
MUKELABAI MUKELABAI

Supervisors: Thibaud Antignac and Gerardo Schneider
Examiner: Regina Hebig

Master's Thesis 2016:NN
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: CCTV street surveillance camera, illustrating one, of several forms of surveillance. *Surveillance video cameras above marina in Gdynia, Paweł Zdziarski, 2007, CC2.5 (reframed)*

Typeset in LaTeX
Gothenburg, Sweden 2016

A Software Architecture to Ensure Surveillance Accountability
MUKELABAI MUKELABAI
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

To meet various security objectives, organisations may employ surveillance technologies such as CCTV cameras or many other forms of online surveillance. However, several concerns have arisen as these technologies are becoming more and more privacy intrusive; thus threatening the civil liberties of the citizens they are meant to protect. More particularly, accountability and transparency are the most endangered privacy principles due to these surveillance activities.

The complexity of surveillance activities and proliferation of personal information in today's ubiquitous computing world renders access control and encryption techniques insufficient to protect privacy. Hence regulations and systems are needed to hold surveillance organisations accountable for the misuse of the information they gather and also make their operations transparent. This requires the use of an approach that ensures public trust and is also acceptable by Surveillance Organizations (SOs) as it should not compromise the main security objectives of the SO. However, some proposed approaches to achieve this accountability are either too weak as they rely on blindly trusting the SO or are too expensive or too intrusive in their requirements which would make them unacceptable by the SO. In certain legal cases, a court of law may request the SO to disclose to it, records related to a citizen under investigation.

This thesis presents an architecture that includes two additional entities to the SO and Court: a Time Stamping Authority and an independent Data Protection Authority (DPA). This is to ensure the accountability of the SO to the DPA and also ensure that the SO can never use any observed fact about a Data Subject (a citizen in this context), in a court of law, without having previously committed that observation to the DPA.The architecture is evaluated by a model of its protocols which are for secrecy, authentication and integrity properties using ProVerif, a well known and mature protocol verification tool. Secrecy is used to prove that a secret observation cannot be leaked thus compromising the SO's mission, while authentication and integrity properties ensure the accountability of the SO.

The results provided by ProVerif show that secrecy and authentication can be preserved thus leading to the conclusion that it is possible for Software Engineers to design architectures that make a surveillance organization accountable while preserving its security objectives.

Keywords: Surveillance, Architecture, Accountability, Transparency, Security, Protocol, Design, Verification.

Acknowledgments

First and foremost, I thank my God who is the source of all wisdom and knowledge and has sustained me through out my period of study.

Next, I would like to thank my supervisors, Gerardo and Thibaud, for their support and guidance throughout this project and my examiner Regina Hebig for her constructive and invaluable feedback.

Lastly but not the least, I thank the **Swedish Institute** through whose financial contribution I have been able to pursue my studies in Sweden, leading to this thesis.

Mukelabai Mukelabai, Gothenburg, June 2016

Contents

List of Figures

List of Tables

1 Introduction

This chapter presents an introduction to the thesis by presenting the problem domain, research objectives and ending with the research methodology utilised.

In the face of security threats, several surveillance systems have been developed in various forms such as airport security checks, CCTV cameras, internet based forms of surveillance, etc. Surveillance is defined as the monitoring of the behaviour, activities, or other changing information (or dynamic states), usually of people—as is our context, for purposes such as influencing, managing, directing, or protecting them [27]. Surveillance tasks can be carried out by private companies, by police services or by intelligence agencies. However, several concerns have arisen as these systems become more and more privacy intrusive; hence threatening the civil liberties of the citizens that they are meant to protect [14].

Privacy is the "claim of individuals, groups, and institutions to determine for themselves when, how, and to what extent information about them is used lawfully and appropriately by others" [47]. The Organisation for Economic Cooperation and Development (OECD), among others such as Canadian PIPEDA and Asian Pacific Economic Cooperation (APEC), presents a core framework for privacy protection through its privacy guidelines found in [34]. One of the privacy principles [35] presented in these guidelines, which is endangered by surveillance, is the accountability principle, which states that a data controller (surveillance organisation in our context) should be accountable for complying with measures which give effect to the other privacy principles stated in the guidelines. Indeed, the need for surveillance organisations to become accountable has recently gained prominence and is becoming a matter of increasing public interest and policy debate worldwide in sectors such as academia, freedom activism and politics, as accountability is being seen as a more desirable mean to protect privacy [34]. For instance, in 2010, the EU Article 29 Working Party on data protection declared the need for surveillance organisations to adopt an accountability principle [36]. In 2013, the US President instituted a Board to review and give recommendations on the operations of the NSA following the leaks by Edward Snowden [14]. Finally, in 2014, the privacy advocate Senator Faulkner of Australia called for "strong and rigorous oversight" over surveillance organisations in order to ensure their "strong and effective accountability" [21].

Accountability is generally defined as the responsibility that an individual or organisation has to someone or for some activity; it is synonymous with answerability. The main components of accountability are transparency, responsibility, assurance and remediation [38]; with transparency being the first step to achieving accountability [37] because it makes wrong acts visible. More precisely, information accountability means that the use of information by an individual or organisation is transparent such that it is possible to determine appropriate and inappropriate use under a given set of rules [47]. A system that provides accountability ensures that individuals and/or organisations would be held accountable for inappropriate use of information.

Public trust and confidence in matters of privacy is lessened as surveillance organisations become more secretive and prolific in their data collection [15, 14]. Thus achieving transparency through accountability of surveillance organisations would regain, to some extent, this lost trust and confidence and would be seen as a first step to protecting the civil liberties of the citizens that these organisations purport to protect [25]. Unfortunately, there lies a challenge in striking a balance between two apparently conflicting goals: meeting security objectives of a surveillance organisation and guaranteeing the privacy of the citizens concerned [14].

1.1 Problem Statement

As already stated, surveillance in its many forms invades privacy and this has led to vast privacy concerns. In many cases, surveillance operations are secretively carried out on data subjects without their knowledge; the justification for this is that it is intrinsic to the nature of these operations and that the operations themselves would be endangered if transparency was brought into the system. Clarke [15] argues that natural defenses against technology driven privacy invasion have proven inadequate. This argument is echoed by Weitzner *et al* [47] who state that access control mechanisms that employ upfront secrecy and information hiding to protect privacy have proven futile. This is because data is increasingly collected through various means and personalised. Storage technology ensures that it is available and database technologies with their analytical power make it discoverable. Hence accountability is now perceived to be the core concept that should underpin mechanisms aimed at protecting privacy [34, 47, 38]. However, even with accountability, "mechanisms must be applied in an intelliegent way, taking context into account and avoiding a 'one size fits all' approach" [38].

We here consider the context of a surveillance organisation that gathers information about data subjects (citizens), which in-turn may be used against the subjects in court cases involving them. In this case, the court issues an order for the SO to disclose some information which may relate to the citizen under investigation, to serve as evidence. The SO then responds by disclosing partial sets of information matching the court order, if any. Generally the court has no way of verifying that the availed information truly matches the SO's internal surveillance activities on the citizen in question and therefore relies on blind trust.

Furthermore, one privacy principle given by the OECD privacy guidelines is the "Individual Participation Principle" which guarantees among other things, that the citizen has the right “to obtain from a data controller, or otherwise, confirmation of whether or not the data controller has data relating to him”. Under current legislations, a citizen has very weak guarantees of gaining access to information about surveillance collections and processes of which he is the data subject (even later, when such a disclosure would no longer defeat the purpose of the collection). Depending on the legal framework, he may make a request to the SO (or a delegated entity) and has to trust the answer without any proof to support it. Even when the SO answers adequately, its answer may be inaccurate or incomplete because of the lack of coherence of its database (which is quite common as depicted by CNIL in France [16]). This lack of transparency generally entails a general lack of public trust and confidence in the work of the SO and undermines the legitimacy of its practices. With accountability as the championed remedy to privacy protection, the challenge still lies in implementing it in a way that would render it acceptable to the SO without compromising the SO’s central mission.

How can we bring transparency through accountability in the way surveillance organisations perform surveillance while providing better privacy guarantees? What methodological approach could Software Engineers follow to build and formally verify architectures that provide better privacy guarantees in the context of surveillance? These are questions this thesis seeks to answer.

1.2 Research Objectives

The following are the null hypotheses driving this study. The negation of each hypothesis is treated as the alternative hypothesis which would be favoured if the corresponding null hypothesis should be falsified.

H1: It is not possible to design an architecture that can be adopted by Software Engineers to preserve privacy through accountability in surveillance contexts.

H2: It is not possible to formally prove properties of this architecture, notably regarding the accountability and the confidentiality aspects. The latter should not compromise the mission of the SO and should prove that the SO cannot disclose to a court an observation that is not registered with the DPA, without being detectable by anybody.

H3: It is not possible to automatically extract, from the model of the protocol, an implementation that makes it possible to develop a proof of concept application.

1.3 Methodology

In this section we first introduce the research methodology employed in our study, namely, Design Science Research [3], and then proceed to describe how we implemented the methodology. Next we present how the requirements for the architecture were elicited and then conclude by presenting the suggested solution based on the requirements, followed by the assumptions we made in designing our solution.

1.3.1 Design Science Research

Design Science Research focuses on *building* and *evaluation* of artifacts designed to meet a specific business need. In the context of Information Systems, Design is considered to be both a process (set of activities) and a product (artifacts) [45]. Design Research "seeks to create innovations that define the ideas, practices, technical capabilities, and products through which the analysis, design, implementation, management, and use of information systems can be effectively and efficiently accomplished"[3]. It has two processes—*build* and *evaluate*, and four artifacts—*constructs*, *models*, *methods* and *instantiations*. The *build* process develops theories and artifacts that meet an identified relevant business need. The *evaluate* process then justifies the utility and efficacy of the built artifacts using methods can be categorised as observational (e.g. case studies), experimental (e.g. controlled experiments), analytical (e.g. formal proofs) or testing (e.g. black-box testing). Figure 1.1, taken from [3], illustrates the concept of Design Science Research. The outcomes of Design

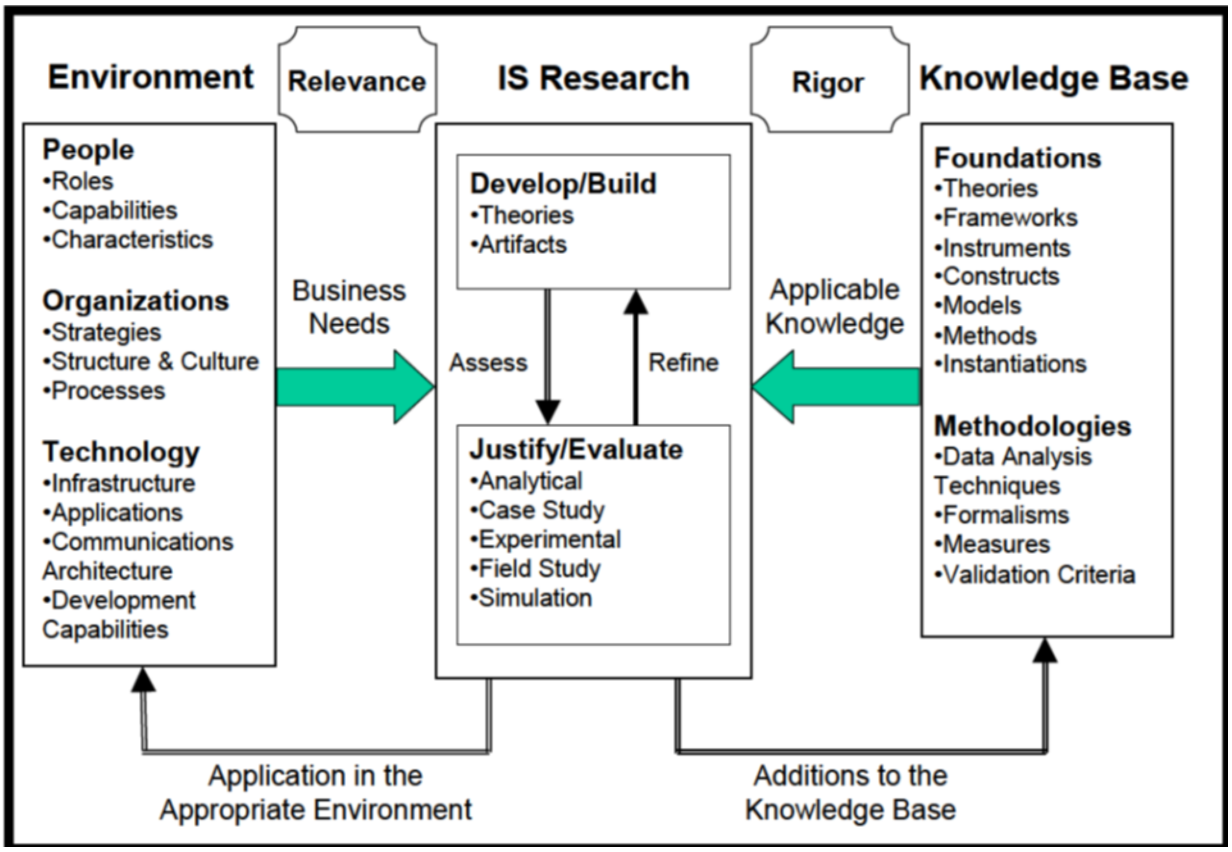


Figure 1.1: Information Systems Research Framework [3].

Research are:

Constructs: Provide a conceptual vocabulary of a problem/solution domain. Constructs arise during the conceptualization of the problem and are refined throughout the design cycle.

Models: Express relationships between constructs to represent a real world situation—the design problem and its solution space.

Methods: Provide guidance on how to solve the problem e.g. a formal mathematical algorithm or textual description of best practice approaches.

Instantiations: Operationalizations constructs, model and methods to show that they can be implemented in a working system.

The following is a summary of the guidelines for conducting design science research as presented in [3]:

Guideline 1: Design as an Artifact —must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.

Guideline 2: Problem Relevance —develop technology-based solutions to important and relevant business problems.

Guideline 3: Design Evaluation —demonstrate rigorously, via well-executed evaluation methods, the utility, quality, and efficacy of a design artifact.

Guideline 4: Research Contributions —provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.

Guideline 5: Research Rigor —apply rigorous methods in both the construction and evaluation of the design artifact.

Guideline 6: Design as a Search Process —search for an effective artifact utilizing available means to reach desired ends while satisfying laws in the problem environment.

Guideline 7: Communication of Research —design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

1.3.2 Thesis Design—Methods and Procedures

Following the procedure of Design Research discussed in section 1.3.1, we answered to each hypothesis (section 1.2) in turn in a constructive way by building artifacts, validating or invalidating them. More specifically:

Architectural Specification: We first made an exploration of the existing framework in the literature and created a conceptual vocabulary of the problem and a solution domain along with expressing the requirements of the stakeholders.

Architectural Design: Next we suggested a solution through the design of a conceptual model and software architecture that expresses relationships between constructs identified in the previous step.

Architectural Evaluation: Since the confidentiality and accountability of sensitive data was of utmost importance, we modeled the sequence diagrams of the architecture as formal protocols and formally verified secrecy and authentication security properties to ensure that they guarantee such properties; this provided some guarantees to the soundness of the architecture with regards to the verified properties. Results of the evaluation were used to suggest architectural security patterns that could be applied to the Architectural Design to achieve the desired security properties.

Implementation Feasibility: Last but not the least, we explored state of the art tools that make it feasible to automatically extract an implementation from the model of the protocol, thus bringing about the realisation of the artifact in an environment.

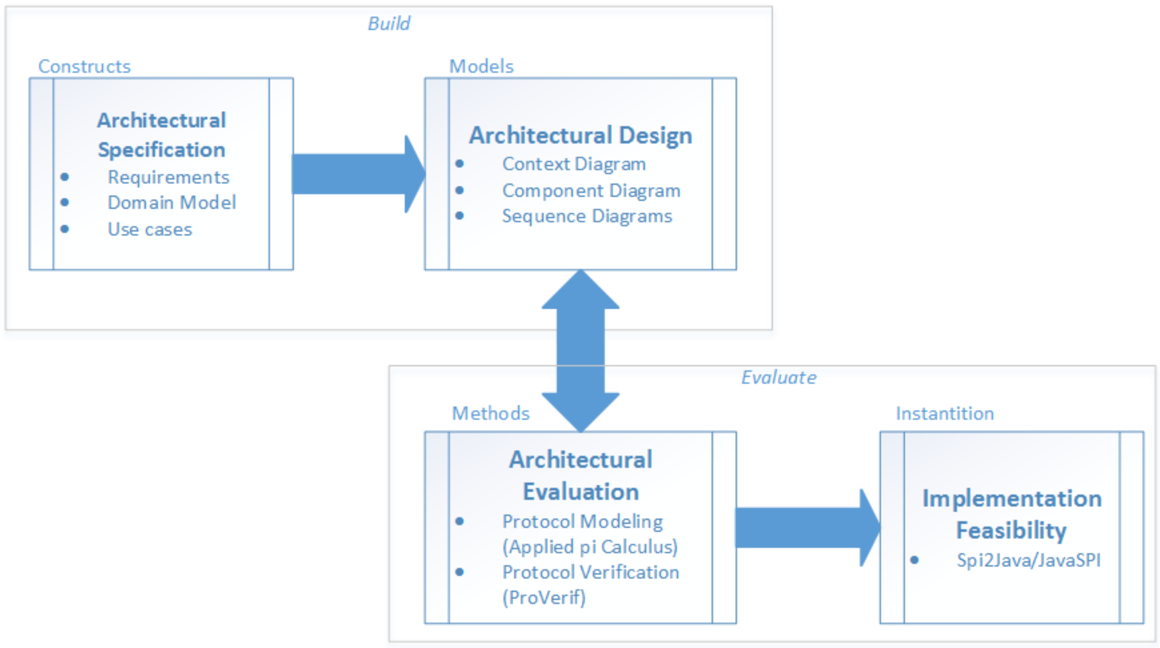


Figure 1.2: Thesis Methodology using the *build* and *evaluate* processes of Design Science Research.

Figure 1.2 provides a summary of the thesis methodology employed in this study. The Architectural Specification and Architectural Design were done in a water fall style. The Architectural Evaluation however was done in a iterative fashion. Each iteration of the evaluation involved the formal modeling and verification of a security property based on the sequence diagrams of the architecture. The verification of each property was performed in a stepwise manner in order to ascertain what attacks could be performed at each stage of the protocol. As may be noticed, we use a double arrow for the relationship between Architectural Design and Architectural Evaluation; this is because the results of the evaluation later changed the architectural design to apply security patterns necessary to enforce the desired security properties.

1.3.2.1 Requirements Elicitation

Requirements for this architecture were derived taking into account some of the recommendations offered by the US President's Review Group [14] on Intelligence and Communications Technologies, appointed in 2013 following the Snowden NSA revelations. From their summary of the top ten recommendations out of fourty six, the following were more instrumental to this thesis:

- Private metadata (e.g. telephone metadata) should be stored, if need be, "by private providers or by a private third party, and which should be available only after an appropriate order by a court."
- Public officials should not have access to private information such as bank records, credit card records, etc. from third parties "without a court order".
- "We need more transparency in the system". Providers must disclose to the public, requests for orders from the government or government should disclose what orders it issued for information that is unclassified.
- "Significant reforms must be adopted to reduce the risks associated with 'insider threats', which can threaten privacy and national security alike. A governing principle is plain: Classified information should be shared only with those who genuinely need to know."

We also drew inspiration from the privacy guidelines [34] and privacy principles [35] presented by the OECD; most notably the Individual Participation Principle and the Accountability Principle ealier mentioned.

1.3.2.2 Proposed Solution

To address H1, we introduced two new components, in addition to the SO and the Court: a Time Stamping Authority (TSA) and a third party Data Protection Authority (DPA). With this proposed architecture, the SO must obtain timestamps for its surveillance observations from the TSA, then using a commitment scheme (see section 3.2.5), send commitments of these observations to the DPA who in turn signifies their receipt by signing them and then retains a copy to be able to service requests from citizens. With the DPA included, the Court can then verify disclosed records from the SO following a court order and Citizens can check with the DPA whether they've been under surveillance before or not.

H2 was addressed by modeling the sequence diagrams of the architecture in H1 as formal protocols in the applied pi calculus and then using the formal models to verify security properties of the protocols in ProVerif [17]. The main security properties verified were *confidentiality* (synonymously referred to as *secrecy* in this thesis) and *authentication*. More details on what these properties involve are given in section 3.2.3. Based on the results of the verification, suggestions were given on what architectural patterns should be implemented on all ends of participating agents in the protocol to ensure that the aforementioned security properties are preserved.

A key aspect in this research is that the results of H2 act as a verification of H1. When a flaw was found in the protocol proposed, a fix was proposed to the previous result. Such negative results obtained during the iterations were documented as they were important insights in the problem as well. Figure 1.3 shows the process followed for modeling the protocol. For each iteration, we considered a requirement (quality or functional). For each chosen requirement, we modeled a solution and then verified it. If the solution passed the verification, we proceeded by adding modeling another requirement provided there were requirements left to model, until all requirements were modeled and verified. However, if the verification failed, we remodeled it as an incremented version which would also be verified until the verification passes. .

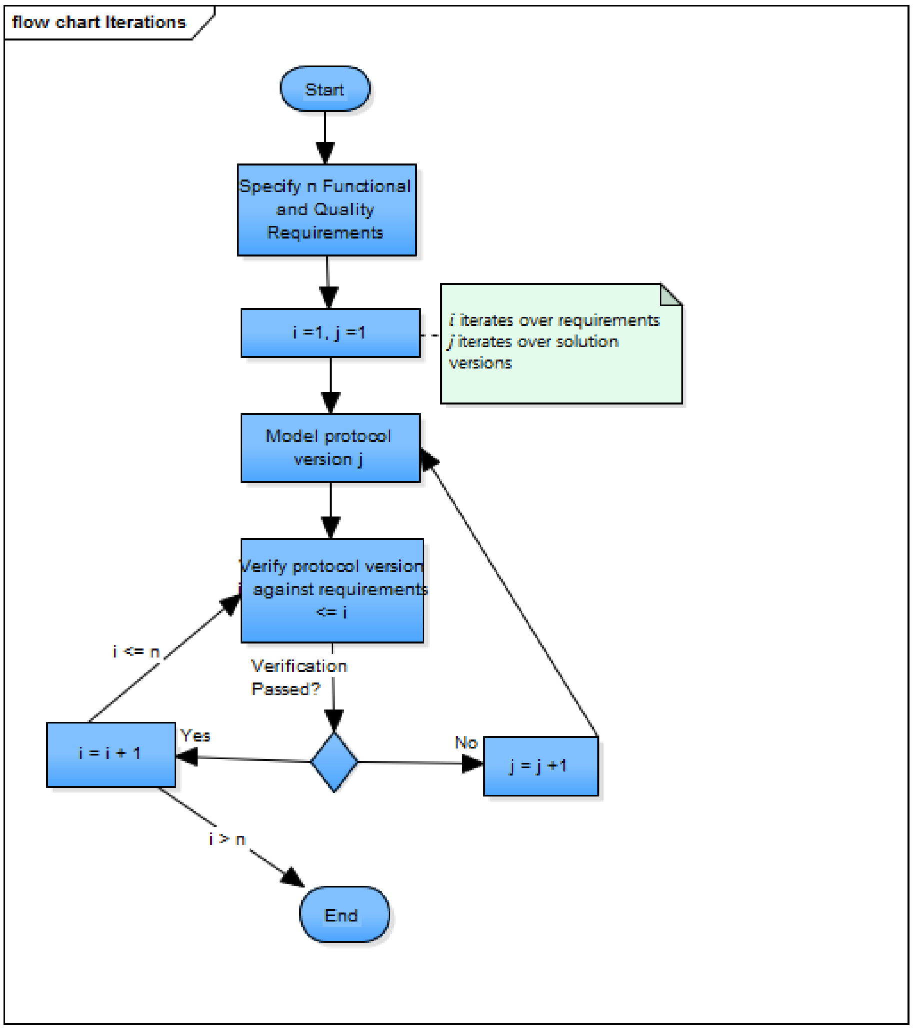

flow chart Iterations
Start
Specify n Functional and Quality Requirements
i =1, j =1
i iterates over requirements
j iterates over solution versions
Model protocol version j
Verify protocol version j against requirements <= i
Verification Passed?
Yes
No
i = i + 1
j = j +1
i <= n
i > n
End

Figure 1.3: Model Iterations

1.3.2.3 Assumptions and Limitations

The following were the assumptions we made in the development of this solution

1. The SO does not falsify or forge observations i.e. whatever is committed to the DPA is considered authentic hence can be used by the court.
2. Protocol considers only verification of evidence based on the records from the SO and not external witnesses brought in by the court.

Before presenting the rest of our contribution, we begin by exploring already existing work in the field through a literature review.

2 Literature Review

The following sections present a review of the literature on surveillance, privacy, and accountability.

2.1 Surveillance and Privacy

As already stated, surveillance is the monitoring of the behaviour, activities, or other changing information (or dynamic states), usually of people for the purpose of influencing, managing, directing, or protecting them [27]. This monitoring is done in a number of ways such as computer surveillance online, telephone systems, CCTV cameras etc. [42] and can be performed by different organisations to serve different purposes such as control, governance, security, and profit. We here consider the kind of surveillance performed on individual citizens as opposed to other kinds of surveillance such as disease surveillance.

As an ambiguous practice, surveillance can have both positive and negative effects: Governments can use surveillance through their various agencies, to control their citizens, monitor threats and possibly prevent criminal activity. On the other hand, this could lead to massive privacy intrusion resulting in a surveillance society without political and personal freedom, and this has been the outcry of civil rights activists and privacy advocates in recent years.

With the proliferation of personal information on the Internet and other media through which surveillance may be conducted, privacy has become more and more increasingly difficult to preserve.

In order to preserve privacy, some democratic governments and institutions around the world employ the use of privacy policies , policy statements and laws, to assure their citizens and clients respectively of some privacy principles such as confidentiality. In cloud computing and other application areas of computer science such as e-commerce, a number of approaches to preserving privacy, presented and reviewed by [32], have been proposed and mainly focus on information hiding by means of access control mechanisms and encryption techniques. Other approaches also reviewed in [32] employ public auditing schemes but still in the area of cloud computing.

However, as [47] has argued, all approaches that rely on upfront secrecy and access control get overwhelmed by the increasingly open information environment and ease with which information can be stored, transported, aggregated and analysed. With the current analytical power, inferences can be made on information which in itself may not be explicitly revealed. Furthermore, [47] argues that approaches based on information hiding and restriction may limit human interaction and are not intuitive to the social nature of our human societies and therefore are misplaced in today's modern ubiquitous information age and therefore a better alternative would be to consider ways of making the *use* of collected data and information *accountable*.

In [28] is presented the SALT framework which is a multidisciplinary approach to preserving privacy in video surveillance systems and "serves as a decision support to assist system designers and other stakeholders in coping with complex privacy requirements in a systematic and methodological way". It "provides reusable, generic and synthetic guidelines, reference information and criteria to be used or modified by experts and other stakeholders and borders on privacy by design and accountability by design." The SALT approach is based on a two step process:

- Guiding surveillance system owners through an assessment process for the legal/socio-contextual and ethical impact of the envisioned system; this includes impact assessment on individuals' privacy.
- During the design phase, designers are referred to socio-contextual, ethical and logical considerations to reduce on impact of the system on individuals' privacy. Accountability features and state of the art privacy preserving technologies are presented them to reduce such impact and ensure transparency.

2.2 Accountability

A number of different possible approaches to bring more transparency and more accountability in organisations have been described in the literature. These appear to lie in a spectrum defined by two extremes: on one end there is pure policy (blind) trust while on the other end there is pure security by deploying trusted mechanisms (balancing the lack of trust in the actor). In our context, approaches based on pure policy would require the citizen or the court to simply trust, without any reservation, the operations of and information supplied by the SO while approaches based on pure security would require secure accountability by the SO, leaving no room for any misbehaving to be undetected at worst, unprevented at best.

One common approach (based entirely on pure policy) drawn from social science research is to require public organisations to disclose their collected data to specific independent review boards or public committees or commissions which, in the case of the SO, would be (only) the courts of law for cases that may require it, the requesting party trusting the information supplied and consuming it as it is. While this approach would be highly welcomed by the SO as it places it under no obligation to comply with any standard or set of rules, the main drawback is it raises several questions regarding "whether or not information made available matches internal organisation activity, whom information is made available to, what sense is made of the information made available and how the information is used" [33].

Another approach would be to use third party auditors who would review the internal operations of the SO and measure them according to “certain principles, expectations standardised measures, benchmarks, performance indicators and so on”. However, this approach may have very low or zero acceptability due to the nature of the SO’s operations. Furthermore, it still falls short of guaranteeing the quality of the data held by the SO.

Finally, one other approach which lies at the pure security end is the use of “policy reasoning tools” suggested by Weitzner et al. [47] which are secure devices that cannot be modified and could be placed at the SO throughout the surveillance system mediating data access and maintaining logs of data transfers. While this is the most trustworthy approach, it would also raise acceptability issues by the SO and hence may not be implementable at all in the context of the SO’s operations.

Pearson [38] suggests that to implement accountability mechanisms in the cloud, both prospective (and proactive) and retrospective (and reactive) accountability approaches may need to be considered. Prospective approaches use preventive tools that prevent an action from continuing to take place or taking place at all (e.g. an access control list). Retrospective approaches use detective controls that permit privacy violations but allow them to be detectable so that corrective measures could be taken. Examples of detective controls are policy-aware transaction logs, language frameworks and reasoning tools (referred to above). In this thesis we consider the retrospective approach and do not try to prevent privacy breaches but rather allow them to be detected.

Drawing from the example of the Fair Credit Reporting Act of 1970 which has successfully protected privacy for more than fourty years, not by limiting the collection of data but by placing strict rules on data usage, Weitzner et al. [47] argue that we gain better accountability by “making better use of the data and by retaining the data that is necessary to hold data users responsible for policy compliance”. This study adopted a similar approach, one in which an independent body would “retain necessary data to hold the SO accountable”. The SO would be required to make commitments of its surveillance observations to an independent organization that we call the Data Protection Authority (DPA). The guarantee that the SO actually behaves as it is required would be assured by a strong incentive which is that it cannot disclose any surveillance observation against a citizen in court unless one that was previously committed to the DPA. Indeed, it would be easy to detect that an observation has not been committed, and then has been hidden by the SO. This would ensure transparency as citizens would be able to check with the DPA for any surveillance operations they may have been subjected to and would also hold the SO accountable by ensuring that any data they use as evidence in court against a citizen is consistent with what was committed to the DPA. At the time of this study, no such an approach had been proposed in the literature.

The next section shall present the necessary backgound theory required to understand the work presented in this thesis.

3
Background Theory

The following sections present design notations used in domain modeling and architectural design, ending with an introduction to protocol verification. The running example demonstrating how to model a security protocol from a given architecture is part of the main contribution to this thesis hence we strongly recommend not skipping referred section as this will make it easier for the reader to follow along with the work presented in later chapters.

3.1 Design Notations and Conventions

This section provides the reader a quick introduction to the notations used to model the domain, use cases, and the architecture. The Unified Modeling Language (UML) [44] is utilised for all the diagrams presented here. Readers having this knowledge may skip this section to the next one giving an introduction to protocol verification (section 3.2).

3.1.1 Domain Modeling

A domain model provides a conceptual vocabulary of a problem and solution domain. Figure 3.1 shows the notations used for domain modeling and the meaning of each relation is detailed in the label attached.

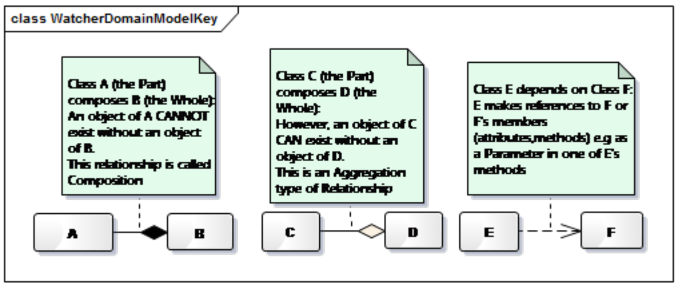

class WatcherDomainModelKey
Class A (the Part) composes B (the Whole): An object of A CANNOT exist without an object of B. This relationship is called Composition
Class C (the Part) composes D (the Whole): However, an object of C CAN exist without an object of D. This is an Aggregation type of Relationship
Class E depends on Class F: E makes references to F or F's members (attributes,methods) e.g as a Parameter in one of E's methods
A
B
C
D
E
F

Figure 3.1: Notations used for Domain Modeling [44]

3.1.2 Use Case Modeling

The required behavior of the system is modeled as use cases which describe interactions between a system and its users (also called actors). Figure 3.2 shows how we model use cases. We distinguish between business use cases and regular (or system) use cases. A business use case shows a business goal to be achieved and the actors that participate to achieve the goal while a system use case shows a single goal the system and an actor should achieve in their interaction.

Another concept introduced is a *misuse case* [41], which describes a function the system should not allow. A misuse case highlights undesired behavior of the system and therefore models the attacks that a system may be faced with. The difference between a use case and misuse case is in the goals: a use case brings value to a system stakeholder while a misuse case is a function which a system shouldn't allow because it is unacceptable and brings loss to one or more stakeholders.

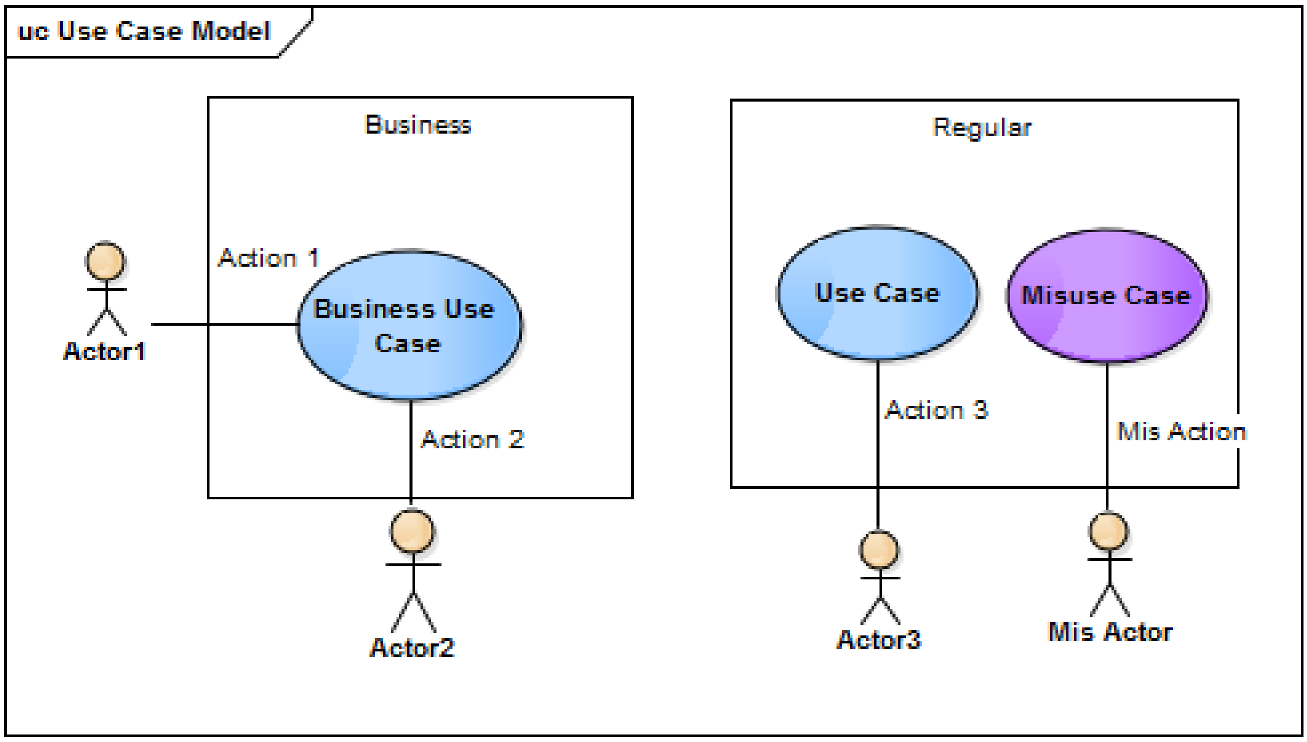


Figure 3.2: Notations used for Use Case Modeling [44, 41]

3.1.3 Architectural Design

Architectural diagrams are designed based on the domain model diagrams and use cases and are modeled using components and interfaces. Components are connected through Interfaces that are service contracts; one component provides a service which another component may require. A provided interface is represented by a hummer while a required interface is represented by a fork. Figure 3.3 illustrates component-interface-component relationships:
We further modeled communication between components as sequence diagrams based on the use cases previously defined. There are three main types of message exchanges: Synchronous, Asynchronous and Self messages. Synchronous messages require the sender to wait for a response from the receiver before proceeding

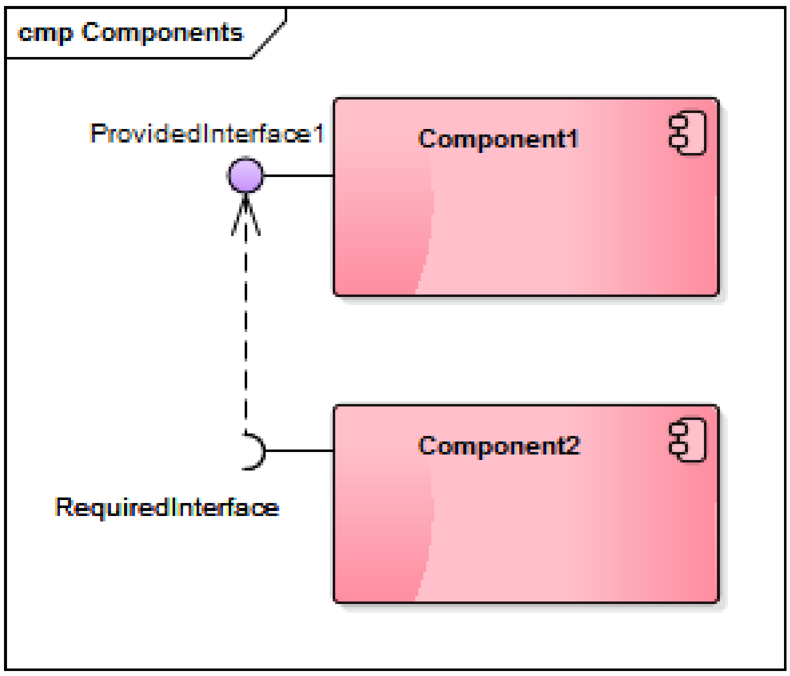


Figure 3.3: Notations used for Component Diagrams [44]

with other tasks. Asynchronous messages do wait for a response. Self messages are internal operations performed by a component. Figure 3.4 illustrates these concepts.

3.2 Protocol Verification

The section presents the reader with an introduction to the knowledge essential to understanding protocol verification. First is presented a discussion of what security protocols are, followed by a discussion of the kinds of communicating agents in protocols, then by a discussion of security properties that are preserved by security protocols. After that is presented a discussion of cryptographic primitives used to model protocols, and then a discussion of how to model protocols using the applied pi calculus and concludes with a presentation of ProVerif which is a protocol verification tool. Throughout this section is presented a running example of a protocol in which two agents communicate; a client and a signature server. The client sends a message to the server to be signed and the server must respond by digitally signing the message and sending it back to the client. Based on this example, all the above discussion of security protocols to verification is covered. It suffices also to mention that this protocol is actually a simplified version to the communication between an SO and the DPA when requesting for a signature for an observation during a commitment. We therefore encourage even experienced readers to skim through the examples presented here for the purpose of familiarization with the work presented afterwards.

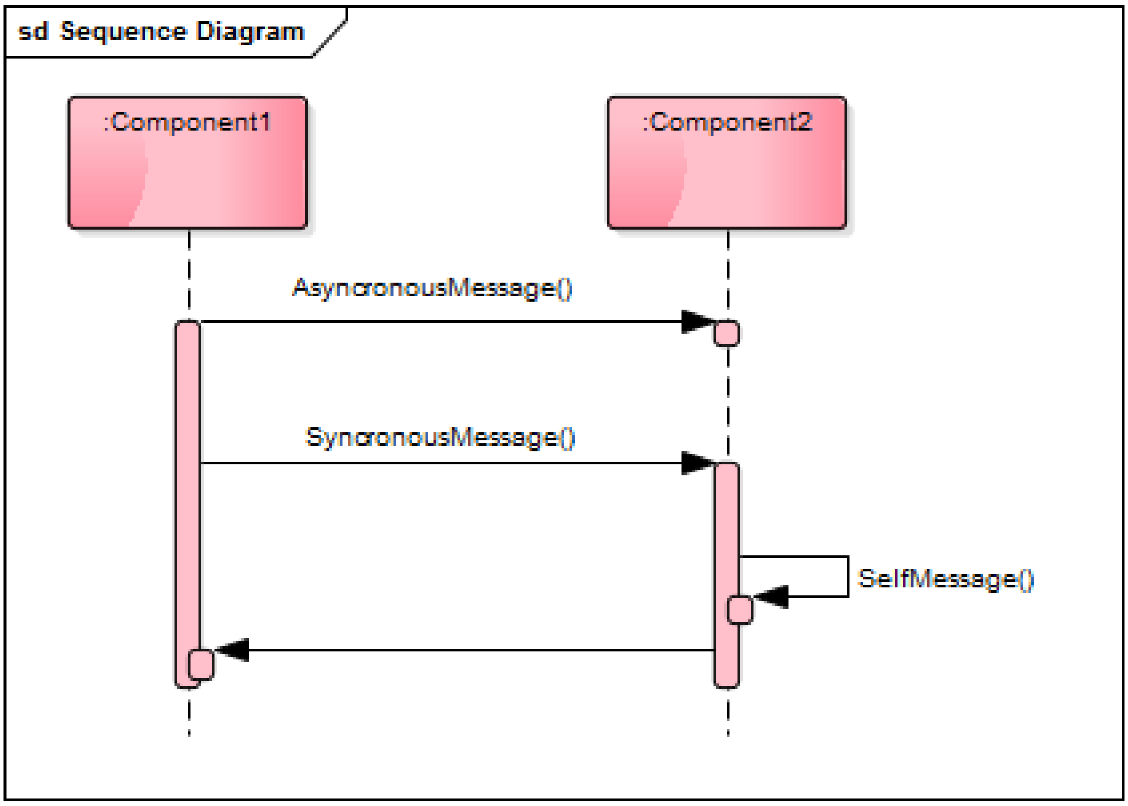


Figure 3.4: Notations Used for Sequence Diagrams [44]

3.2.1 Security Protocols

What are Security Protocols?

In Computer science, a communication protocol is a set of rules that govern end to end telecommunications between two agents (also known as participants or principals or entities). The protocol provides a specification for the interactions between the communicating agents. Because of the distributed nature of this communication, security protocols are therefore distributed and concurrent programs that secure communication by means of cryptographic techniques such as encryption to ensure security properties such as confidentiality of data [17]. Examples of communication protocols include the HTTP protocol that governs exchange of data in HTML format and the SMTP that governs the exchange of email. An example of a security protocol is the SSL protocol that underlies the https protocol in web browsers and performs functions such as encrypting web search queries between a host and a search engine.

Why do we Need to Verify Them?

Security protocols need to be verified because, unlike other safety critical systems, "properties of security protocols must hold in the presence of an arbitrary adversary". Any cases of failure or design flaws in such protocols can have huge financial and societal impact. In an empirical study conducted by Cavusoglu et al in 2004

[13] on the effect of internet security breach announcements on market value, it was observed that a breached firm lost about 2.1 percent of its market value within two days of the announcement, which translates to a market capitalization loss of \$1.65 billion dollars per breach. In his online article of April 2015, Howarth [23] indicates that the top factors for calculating monetary loss resulting from security breaches affecting financial institutions were customer reimbursements, and audit and consulting services and that the New York State Department of Financial Services found the deployment of additional security measures as only the third most costly impact of cybersecurity breaches. Other institutions considered additional factors such as reputational damage, though difficult to quantity. All these studies and many other instances of security breaches provide a stronger incentive for security protocol verification.

3.2.2 Channels and Agents

Protocols use channels to allow agents to communicate; these channels may be a public network such as the internet and the agents maybe two computers (or persons) exchanging messages. Communication protocols usually assume trusted channels and honest agents where as security protocols assume untrusted channels and dishonest (or hostile) agents.

Trusted Channels and Trusted Agents

In a trusted channel, no hostile agents are able to access the medium of communication in order to interfere with the protocol. Trusted agents are those that cooperate to achieve the goal of the protocol.

Untrusted Channels and Dishonest Agents

An untrusted channel is one in which hostile agents access the medium of communication to subvert the protocol by means such as reading, modifying, injecting messages and manipulating messages. An example of an untrusted channel is the internet. A dishonest agent is one that acts as a regular participant of a protocol but actually subverts the rules of the protocol to his own advantage. An example of a dishonest agent may be an e-commerce seller that falsely denies receiving payment from a client. An attacker is any hostile agent that is either a dishonest agent or an outsider subverting a protocol through an untrusted channel.

3.2.3 Security Properties

The goal of any security protocol is to ensure that one or more security properties [17] are preserved. The kind of security properties preserved by a protocol depends on the purpose of the protocol and context. We present here informal definitions of some properties, namely Secrecy, Authentication, Integrity, Anonymity, Unlinkability, Non repudiation and Fairness. However this thesis focuses on *Secrecy*, *Authentication* and *Integrity*; the latter four are given for the purpose of letting the reader know of some other existing security properties apart from those we address.

1. **Secrecy**
   Secrecy refers to the prevention of unauthorised disclosure of information and has two flavours: weak secrecy and strong secrecy.

   Weak secrecy focuses on *reachability* and means that an attacker cannot deduce the contents of a message by reaching a state in a protocol run where he has knowledge of a secret. This means that given a secret s, and an attacker's current knowledge, he should not reach a state where he can discover the secret s based on his current knowledge of the protocol.

   Strong secrecy focuses on *indistinguishability* and refers to the fact that an attacker should not be able to deduce any information about the messages communicated in a protocol run; this deducible information includes, but is not limited to, the length of messages, and whether or not the same message has been sent twice.

   The difference between weak secrecy and strong secrecy is that, with weak secrecy it is possible for an attacker to see the difference when the value of a secret changes, where as it is not the case with strong secrecy [10]. For instance, "when a process encrypts a message m, an attacker can differentiate between different messages since their ciphertexts will be different" but if strong secrecy techniques are used such as probabilistic encryption, the randomness in the encryption would yield different ciphertexts for the same value of m, hence m would be a strong secret.

2. **Authentication**
   Authentication is among the most important security properties and focuses on verifying identities of communicating entities (agents) or messages exchanged. Two of its forms are entity authentication and message authentication. Entity authentication aims at verifying the identity of an entity; that an attacker does not impersonate an entity A when communicating with an entity B. Message authentication aims at verifying that a message comes from the agent it claims to come from; we achieve this through digital signatures.

   Closely related to Authentication is the notion of correspondence properties introduced by Woo and Lam [49]. A correspondence property states that if an event e has happened, then an event e' must have happened before. In the context of a protocol run, we say that if an event e is "B accepts a run of the protocol", then an event e' must have happened before which is, "A started the run of the protocol". Four correspondence properties are presented here, which are Aliveness, Weak Agreement, Non-injective Agreement and Injective Agreement:

   Aliveness: Aliveness is the weakest form of authentication which requires that whenever a honest agent A completes a run of a protocol apparently with another honest agent B, then B has previously run the protocol. This property fails to capture some attacks such as identity impersonation.

Weak Agreement: In addition to aliveness, this property requires that the agents agree on their identities.

Non-injective Agreement: Sometimes agreeing on identities may not be sufficient if we want the agents to agree on some other messages hence non- injective agreement means that in addition to weak agreement, the agents run the protocol with the same data set. However this could still suffer from replay attacks.

Injective Agreement: injective agreement adds to non-injective agreement by ensuring that every run of a protocol by A corresponds to exactly one unique run by B. For instance, for each deposit of cash into a bank account by a client,the bank should credit the client's account only once.

This thesis utilises injective agreement.

3. **Integrity**
   Integrity prevents unauthorized modification of information. In this thesis, integrity is ensured through digital signatures which also provide non-repudiation.

Other Security Properties

The following are a few more security properties, some of which are addressed implicitly and others are not addressed in this thesis but may be of interest to the reader.

1. **Anonymity**
   This property aims at prevention of identification of specific properties of individual events from a set of events. Examples of applications requiring anonymity are e-voting applications. However, anonymity is incompatible with authentication hence we did not consider it in our study.

2. **Unlinkabibility**
   Unlinkability is very much used in Radio Frequency Identification (RFID) systems where it might be desirable not to allow an attacker to link several sessions i.e. to infer that the sessions involve a same user.

3. **Non-repudiation**
   Non-repudiation prevents an agent from falsely denying responsibility for their actions; for instance, a sender of a message should not falsely deny having sent the message. This property is implicit in didital signatures, e.g. the DPA cannot deny having signed a message.

4. **Fairness**

Fairness prevents one participant from gaining advantage over another by aborting the protocol; for instance one participant pays for merchandise and the other doesn't send the merchandise or vice versa.

3.2.4 Cryptographic Primitives

Security protocols make use of cryptographic primitives which include the following:

- Symmetric and asymmetric encryption
- Digital signatures
- Cryptographic hash functions
- Message authentication codes (MACs) also known as keyed hash functions
- Random number generation

Symmetric and Asymmetric Encryption

When plaintext is encrypted, it is called ciphertext. *Symmetric encryption* uses the same cryptographic keys for the encryption of plaintext and decryption of ciphertext. *Asymmetric encryption* on the other hand uses two kinds of keys: a *public key* shared widely and a *private key* known only by the owner. Using this public key system, anyone can encrypt a message with the public key of the receiver but only the receiver can decrypt it with his private key hence in asymmetric encryption, we keep secret only the private key while the public key is shared without compromising security [48].
The following are the notations we use for symmetric encryption and asymmetric encryption given a message *m*, private key *k* and the public key of *k* as *pk(k)*; we assume that keys are unguessable:

Symmetric encryption:

- $senc(m, k)$ —encrypts *m* using key *k*.
- $sdec(senc(m, k), k)$—*sdec(...)* is a decryption function that gives *m*, given its encryption key *k*

Asymmetric encryption:

- $aenc(m, pk(k))$—encrypts *m* with $pk(k)$.
- $adec(aenc(m, pk(k)), k)$—*adec(...)* is a decryption function that gives *m*, given that the private key of its encryption is *k*.

Digital Signatures

A digital signature is like a handwritten signature, it provides authenticity for a message and ensures that the sender of the message cannot deny having sent the message (non-repudiation) and also provides some proof that the message wasn't modified in transit (integrity). Digital signatures use asymmetric cryptography as described above. The notation we use for a digital signature given a message *m*, private signing key *k* and public signing key of *k* as *spk(k)*:

sign(m,spk(k))

More appropriate functions for retrieving signed messages and verifying signatures are discussed in later sections.

Cryptographic Hash Functions

A cryptographic hash function is a one way hash function that makes it computationally impossible to recreate its input data from its hash value. Cryptographic hash functions are used in digital signatures and message authentication codes but are not limited to these applications. Given a message *m*, its hash is denoted as:

- *hash(m)*

Message Authentication Codes (MACs)

A message authentication code is a piece of information used to provide the authenticity and integrity of a message. However, since message authentication is implicit in digital signatures, MACs are not useful in this study.

Random Number Generation

A *nonce* is a number used once in a cryptographic communication and is usually a random or pseudo-random number that may serve purposes such as being a session key. In protocols implementing authentication, nonces are used for things such as preventing old communications from being used in replay attacks by guaranteeing for instance the uniqueness of a session. A nonce is denoted as *N*.

If we have two participants *A* and *B*, each generating their own nonces, then we denote their nonces as *Na*, and *Nb* respectively. More generally, we suffix the identity of an agent or any letter of our choosing to the letter N to denote a nonce generated by that agent.

In summary, this thesis employs the following cryptographic primitives: symmetric and asymmetric encryption, digital signatures, hash functions and random numbers (nonces).

3.2.5 Commitment Scheme

A commitment scheme [22] is a cryptographic technique that allows one party to commit to a chosen value in a protocol while keeping it hidden from others, but with the ability to reveal the hidden value at a later point in time. It has two important properties that must be preserved: the *hiding* property and the *binding* property. The *hiding* property means that the receiver cannot know the value of the secret message until revealed by the sender, while the *binding* property means that the secret message must be bound to exactly one unlocking message called the *opening*.

The two phases of a commitment scheme are the *commit* phase, and the *reveal* phase.

The *commit* phase in some protocols involves the sender sending a single message, called the *commitment*, to the receiver. The receiver should not know the specific value of the message (*hiding* property). At some later point during the *reveal* phase, the sender can send an *opening* message which would allow the receiver to check the value of the original hidden message; this works if the original message can be bound to only one *opening* message (binding property).

Commitment schemes are used in applications such as: coin flipping [31]—that allows dispute resolution through coin flipping, and digital signature schemes—that allow publishing of verifiable hashes of data. This thesis employs a commitment scheme in the context of a signature scheme to allow the SO to commit observations to the DPA. However, in order for the DPA to be able to respond to requests from citizens, some data about the observations e.g. identifiers shall not be hidden from the DPA.

3.2.6 Protocol Modeling

Material presented in this section gets inspiration from a tutorial [17] by Cortier and Kremer. However, it is adapted to the protocol contributed in this thesis.

Protocols are implemented in programming languages like C++, C or Java; however, protocol verification is an instance of formal verification which is performed on abstract mathematical models of protocols. Abstract models are used because they provide only the details relevant for the proof unlike concrete programs. The result of protocol verification is a formal proof that provides the correctness of the protocol model with regard to a formal specification or property.

Different symbolic models are used to represent and reason about protocols and these include process algebra (for instance applied pi calculus [1]), strand spaces [20], constraint systems [29] and Horn clauses [9]. Though differing in many aspects, these models all represent protocol messages by terms. Precise details or values of nonces, keys or identities are abstracted away leaving only the structure of the message which is modeled as a special labeled graph called a term.

Terms

As already shown in section 3.2.4, cryptographic primitives are represented by function symbols where a function symbol f has an associated arity (number of arguments). A finite set of function symbols is called a *signature* (not to be confused with a digital signature). *Variables* are used to represent unspecified parts of messages. *Names* represent atomic data such as *identities*, *nonces* and *keys*. A standard signature in the context of security protocols is a set of *constructor* function symbols and is represented as

F_{std} = *{senc,aenc,pair,pk}*

where *senc, aenc* and *pair* are symbols of arity 2 representing respectively symmetric encryption, asymmetric encryption, and concatenation, whereas *pk* is a symbol of arity 1, representing the public key associated to some private key. The corresponding signature for *destructors* is given as:

F_{dec}= *{sdec,adec,fst,snd}*

corresponding to, respectively, symmetric decryption, asymmetric decryption, first, and second projections on a pair.

The set of terms of the signature *F*, the variables *X*, and the names *N* is denoted as $T(F, X, N)$ which is defined as names, variables and function symbols applied to other terms. Given F_0 to be an arbitrary finite set of constant symbols and given a term algebra $T(F_{std} \cup F_{dec} \cup F_0 \cup X)$, the properties of concatenation and standard symmetric and asymmetric encryption can be modeled by the following:

$sdec(senc(x, y), y) = x \qquad adec(aenc(x, pk(y)), y) = x$

$fst(pair(x,y)) = x \qquad snd(pair(x, y)) = y$

fst is a projection on the first term of a pair and *snd* is a projection on the second term of a pair.

Assumptions on Perfect Cryptography

Protocol verification assumes perfect cryptography and focuses on the correctness of the protocol rather than the cryptography; hence the following assumptions are made about the utilized cryptography:

- One cannot learn anything about or modify an encrypted message unless one has the right key
- Keys cannot be guessed from encrypted text
- Random numbers cannot be guessed
- Hashes are one way and collision free; one way meaning that a hashed message cannot have its value retrieved, and collision free meaning that two different messages should have two unique hashes.

Attacker model

We also assume that the public channel used for communication in a protocol is controlled by an environment that captures the attacker capabilities given by Dolev-Yao [19]: the attacker can read, modify, delete or inject messages and also manipulate messages. In particular, the Doley-Yao inference system states that:

- The attacker can concatenate terms and retrieve terms from a concatenation i.e. given *x, y* as terms, one can concatenate them as *(x,y)* and given the previous concatenation, once can retrieve its terms.
- The attacker can encrypt and decrypt symmetrically given the corresponding

key i.e. given *x*, and *y* as terms, one can *senc(x,y)* and given the previous symmetric encryption and a key *y*, once can retrieve *x*.

- Similarly, the attacker can encrypt and decrypt asymmetrically given corresponding public and private keys i.e. given *x*, and *y* as terms, one can *aenc(x,y)* and given an encryption *aenc(x,pk(y))* and a private key *y*, one can retrieve *x*.

Authentication of agents: Needham Schroeder Protocol

As stated earlier, authentication aims at verifying identities of communicating agents to ensure that no honest agent is impersonated by an attacker. To authenticate agents, we use the corrected Needham-Schroeder protocol also known as Needham-Schroeder-Lowe protocol [26]. The following is the basic NS protocol

1. A —> B: aenc((A,Na),pkB)
2. B —> A: aenc((Na,Nb),pkA)
3. A —> B: aenc(Nb,pkB)

Two agents, *A* and *B*, want to authenticate each other by engaging in a challenge response before they further communicate. *A* sends an encrypted pair of its identity *A* and a nonce *Na* to *B*, encrypted with *B*'s public key. *B* responds by creating a new nonce *Nb*, pairs it with the nonce received from *A* and sends it back to *A* by encrypting it with the public key of *A*. *A* would check whether the first projection of the pair received corresponds to the nonce earlier sent, which is *Na* and if so, then sends back *B*'s nonce *Nb*. *B* would in turn also check whether the nonce received corresponds to the earlier one created which is *Nb*. At the end of this run of the protocol, *A* would know that it was truly communicating with *B*, and *B* would also know that it was communicating with *A* and the secrecy of the nonces would be preserved.

However, Lowe[26] discovered the man in the middle attack in which an attacker could impersonate *A* in its communication with B. Let C be an attacker who impersonates A:

A —> C:	aenc((A,Na),pkC)
C(A) —> B:	aenc((A,Na),pkB)
B —> C(A):	aenc((Na,Nb),pkA)
C —> A:	aenc((Na,Nb),pkA)
A —> C:	aenc(Nb,pkC)
C(A) —> B:	aenc(Nb,pkB)

First, *A* sends its identity and its nonce to a dishonest agent *C*, supposing it to be *B*. *C* then forwards this to *B* which would respond by sending the two nonces encrypted with *A*'s public key. Since *C* can't decrypt this pair, it forwards it to *A*, at which point *A* responds by sending back the nonce *Nb* to *C*, again supposing it to be *B*. At this point *C* knows the nonce *Nb* and then forwards the message to *B*. Lowe fixed this protocol by letting *B* include its identity in the message sent back to *A*. This way, *A* is able to check if the message received is truly from *B* or not.

The resulting protocol is:

A —> B:	aenc((A,Na),pkB)
B —> A:	aenc((Na,Nb,B),pkA)
A —> B:	aenc(Nb,pkB)

At the end of this authentication, both *A* and *B* are certain that they are communicating to each other and that the nonces *Na* and *Nb* are secret.

Modeling a protocol

A protocol is modeled based on the interactions of the communication that takes place between two agents. In software engineering, since the sequence diagrams of an architecture provide the interactions between entities, this would be a good place to start. For each interaction, we must then consider what security properties we want to preserve and what could go wrong and then apply appropriate cryptographic techniques.

Let us consider a simple hypothetical use case of a *client A* requesting for a digital signature for a secret message *m* from a *server B*. The following would be the use case:

Name: Request signature for message
Main success scenario:

1. A sends message m to B
2. B signs m
3. B sends signed m to A

The corresponding sequence diagram would be what is presented in figure 3.5:

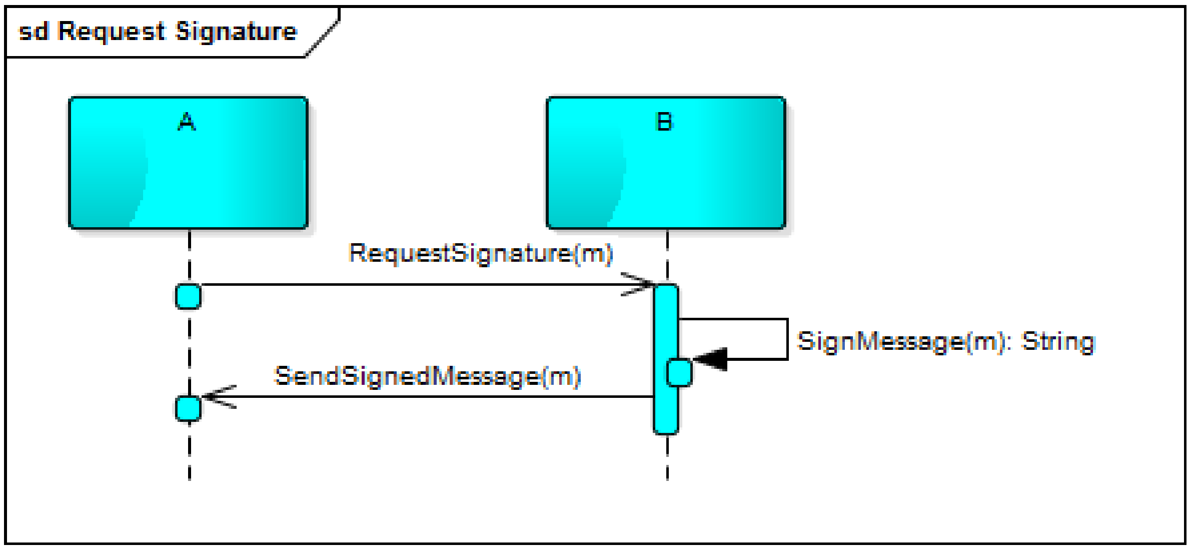


Figure 3.5: Sequence Diagram for Use Case: Request Signature

From the given sequence diagram in figure 3.5, we then write an informal description of the protocol as follows:

> *A —> B: m*
> *B —> A: sign(m,sskB)*

The notation A —> B: *m* indicates that *A* sends message *m* to *B*.

The notation B —> A: *sign(m,sskB)* indicates that *B* sends a pair of a message; *m* signed with the signing secret key of *B—sskB*.

From this simple description, we then begin to reason about which security properties we want to preserve at each point of the protocol. We ask ourselves what could go wrong at each step. However, certain attacks (bugs) in the protocol may sometimes not be obvious and that is where automated tools become of assistance, but we will cover them later. By careful examination of the protocol above, one will notice that it is possible for an attacker to get hold of *m* since he controls the channel of communication; it is also possible for an attacker to impersonate the *server B* and give a wrong signature or impersonate the *client A* and steal the signature meant for *A*. Therefore, we should consider properties such as the *secrecy* of *m*, and the *authentication* of the agents etc. This would mean that we have to modify our protocol to ensure that these properties are preserved i.e. *m* is not leaked to an attacker as it should be secret and that none of the agents is impersonated.

Next, to preserve the secrecy of *m,* we will introduce asymmetric encryption.

Let *pkA, pkB* be the public keys of both *A* and *B* respectively. The protocol now becomes:

> *A —> B: aenc(m,pkB)*
> *B —> A: aenc(sign(m,sskB),pkA)*

We now have a protocol that to some extent preserves the secrecy of *m* by encrypting any message that transmits *m.* When sending *m* to *B* for signing, *A* encrypts it with the public key of *B* so that only *B* should be able to decrypt it and also when sending the signed message to *A, B* encrypts it with the public key of *A* so that only *A* should be able to decrypt it. Notice that what is encrypted by *B* is the signed message.

To preserve *authentication* we would have to proceed in a similar fashion as above by ensuring that *identities* of both entities are sent together with the messages exchanged. This means that instead of *A* only sending *m*, it should include its identity which we can assume to be its *public key* and similarly for *B*. The following would be the resulting description:

> *A —> B: aenc((m,pkA),pkB)*

B —> A: aenc((sign(m,sskB),pkB),pkA)

This would allow A to verify the identity of the server B and vice versa if need be.

The above abstract notation, while convenient to explain a protocol model, does not completely model the protocol because it has ambiguities and leaves out many aspects, for instance, concurrency. The next step therefore is to translate it into the applied pi calculus [1] which would in turn be used by automated tools for protocol verification. The next section covers the basics of the applied pi calculus necessary for the reader to be able to follow the study.

3.2.7 Applied Pi Calculus

This discussion of the applied pi calculus is also based on a tutorial [17] by Cortier and Kremer.

The applied pi calculus represents protocols as processes and has two kinds of processes: plain and extended processes.

Plain processes are generated by the grammar in figure 3.6.

$P,\ Q,\ R$	:= Plain processes
	0
	$P \mid\mid Q$
	$!P$
	$\nu n.P$
	if $t_1 = t_2$ then P else Q
	in$(u, x).P$
	out$(u, t).P$

Figure 3.6: Syntax: Plain Process

"$t_1, t_2, ...$ range over terms, n over names, x over variables and u is a meta-variable that stands for either a name or a variable of *channel* type. The 0 process is the process that does nothing. Parallel composition $P \mid\mid Q$ models that processes P and Q are executed in parallel. The replication of P, denoted $!P$, allows an unbounded number of copies of P to be spawned. New names are created using the new operator νn, which acts as a binder and generates a restricted name. The conditional if $t_1 = t_2$ then P else Q behaves as P whenever $t_1 =_E t_2$ and as Q otherwise.The statement $t_1 =_E t_2$ means that the two terms are equal based on some equational theory. Finally, in*(u, x).P* expects an input on channel u that is bound to variable x in P and out*(u,M).P* outputs term M on channel u and then behaves as P" [17].

Extended processes are generated by the grammar in figure 3.7:

$$\begin{array}{ll} A,\ B,\ C & := \text{Extended processes} \\ & P \\ & A \mid\mid B \\ & \nu n.A \\ & \nu x.A \\ & \{t/x\} \end{array}$$

Figure 3.7: Syntax: Extended Process

They extend plain processes by active substitutions, and allow restrictions on both names and variables. An active substitution $\{t/x\}$ allows processes to address a term by a variable. The scope of this access may be restricted using the ν operator on variables. This also allows to define local variables as follows: the construct let $x = t$ in P is defined as $\nu x.(P \mid\mid \{t/x\})$. When the variable x is not restricted, it means that the environment, which represents the attacker, may use x to access the term t.

Modeling a Protocol as a Process

Using applied pi calculus we shall now show how to model a security protocol as a process or processes. Let's take for instance the protocol we discussed earlier called *Request Signature* whose informal description is the following:

A —> B: aenc((m,pkA),pkB)
B —> A: aenc((sign(m,sskB),pkB),pkA)

Each agent in the protocol plays a *role* and it is the roles that we shall model as processes. For each role, we shall instantiate the corresponding process with the required keys e.g. *A* needs to have a private key *skA* and the public key of *B, pkB, B* needs the private signing key, *sskB* and the private encryption key *skB*. Using the signatures F_{std} and F_{dec}, we model the processes as follows:

The process for the role of *A* is modeled as follows:

$$\begin{array}{lll} P_A(skA, pkB) & = & \nu m.out(c, aenc((m, pk(skA), pkB)). \\ & & \mathrm{in}(c, x). \\ & & 0. \end{array}$$

Process *A* first creates a fresh and restricted message *m* and then outputs it on the public channel *c*. The message is concatenated with the client's identity— the public key associated with the private key *skA*, and then the pair is encrypted with the public key of the server *B*. The client *A* then begins to wait for input, which as expected, should be the signed message *m*. The fact that *m* is restricted, means that it is initially unknown to an attacker.

The process for *B* is modeled as follows:

$$\begin{array}{lcl} P_B(sskB, skB) & = & \text{in}(c, y). \\ & & \text{let } ym = \mathit{fst(adec(y,skB))} \text{ in} \\ & & \text{let } pkY = snd(adec(y, skB)) \text{ in} \\ & & \text{out}(c, aenc((sign(ym, spk(sskB)), pk(skB)), pkY) \\ & & 0. \end{array}$$

Assuming we have a function symbol *spk(k)* that returns the public key associated with a signing secret key k, the server process B only needs to be instantiated with its secret signing key *sskB* and its secret encryption key *skB*. It first begins by waiting for input on the public channel c; intuitively this should be a message to be signed. Once a message is received, it is decrypted using B's secret key. The received message will be a pair containing first the message to be signed and then the public key of the requesting client. Hence first (*fst*) and second (*snd*) projections are made on the pair and the message to be signed is saved in a variable ym while the public key of the requesting client is saved in the variable pkY respectively. B then signs ym with its public signing key associated with the secret signing key *sskB* and then pairs the signed message with its identity which is the public key associated with its private encryption key *skB*, then encrypts the pair with the received public key of the requesting client pkY and then outputs the message on the public channel c.

The last phase in the model is to put the processes together into a single process that actually runs the two processes. We shall call this process P^n_{rs} where *rs* stands for the name of our protocol—Request Signature, and n is the version number as we shall modify it a number of times.

$$\begin{array}{lcl} P^1_{rs} & = & \nu skA, skB, sskB.(P_A(skA, pk(skB)) \mid\mid P_B(sskB, skB) \mid\mid \\ & & \text{out}(c, pk(skA)) \mid\mid \text{out}(c, pk(skB)) \mid\mid \text{out}(c, spk(sskB))) \end{array}$$

At the very top level of the main process above, we create private encryption keys *skA*, *skB* and the private signing key *sskB*. We then instantiate P_A and P_B and let them run in parallel. This shows that the agent identified by *pk(skA)* is executing an instance of the role P_A with the agent identified by *pk(skB)*.We also output the public keys of the previously created secret keys on the public channel c and make them available to the attacker.

This model however, fails to capture Lowe's man in the middle attack [26] since it does not include any dishonest agent C. Let us assume that the attacker posses a secret key *skC*, we shall modify the previous model and include an instance of P_A which is instantiated with the *pk(skC)* to indicate that A could start a session with a dishonest agent C.

$$\begin{array}{lcl} P^2_{rs} & = & \nu skA, skB, sskB.(P_A(skA, pk(skB)) \mid\mid P_A(skA, pk(skC)) \mid\mid \\ & & P_B(sskB, skB) \mid\mid \text{out}(c, pk(skA)) \mid\mid \text{out}(c, pk(skB)) \mid\mid \text{out}(c, spk(sskB))) \end{array}$$

The second version above definitely captures the man in the middle attack; however, one does not know a priori with whom agents should start a session. We therefore leave it to the attacker to decide and instead of explicitly adding an instance of P_A starting a session with *pk(skC)*, we include an input that is used to define the public key given to the client who has the initiator role. We let P_A read a public key from the public channel and then start a session with that public key; we call the public key *xpk*.

$$\begin{aligned} P^3_{rs} &= \nu skA, skB, sskB.(\mathrm{in}(c, xpk).P_A(skA, xpk) \mid\mid P_B(sskB, skB) \mid\mid \\ &\quad \mathrm{out}(c, pk(skA)) \mid\mid \mathrm{out}(c, pk(skB)) \mid\mid \mathrm{out}(c, spk(sskB))) \end{aligned}$$

This ensures that the attacker can start a run of the protocol with any public key available including his. However, version 3 still only captures a single session per role and this may lead our model to miss out existing attacks because many attacks require several parallel sessions of the same role. To resolve this, we include replication (!).

$$\begin{aligned} P^4_{rs} &= \nu skA, skB, sskB.(!\mathrm{in}(c, xpk).P_A(skA, xpk) \mid\mid !P_B(sskB, skB) \mid\mid \\ &\quad \mathrm{out}(c, pk(skA)) \mid\mid \mathrm{out}(c, pk(skB)) \mid\mid \mathrm{out}(c, spk(sskB))) \end{aligned}$$

Version 4 allows for multiple arbitrary sessions to be executed by both A and B. However, it is still possible that both roles P_A and P_B could be executed by the same agent. Moreover, this model only allows two honest agents executing several sessions and yet an attack may require several agents executing several sessions. We therefore add replication to allow the model to create an arbitrary number of honest keys, each of which could be used in an arbitrary number of sessions. We also allow both agents to play both roles by interchanging their public keys.

$$\begin{aligned} P^5_{rs} &= !\nu skA, skB, sskB.(!\mathrm{in}(c, xpk).P_A(skA, xpk) \mid\mid !P_B(sskB, skB) \mid\mid \\ &\quad !\mathrm{in}(c, xpk).P_A(skB, xpk) \mid\mid !P_B(sskB, skA) \mid\mid \\ &\quad \mathrm{out}(c, pk(skA)) \mid\mid \mathrm{out}(c, pk(skB)) \mid\mid \mathrm{out}(c, spk(sskB))) \end{aligned}$$

We finally summarise it as follows to allow for the symmetric nature of the roles expressed in version 5 above:

$$\begin{aligned} P^6_{rs} &= !\nu sk, sskB.(!\mathrm{in}(c, xpk).P_A(sk, xpk) \mid\mid !P_B(sskB, sk) \mid\mid \\ &\quad \mathrm{out}(c, pk(sk)) \mid\mid \mathrm{out}(c, spk(sskB))) \end{aligned}$$

3.2.8 ProVerif

For any protocol verification tool, the following three properties are desirable even though not all may be guaranteed in certain cases [17]:

1. *Soundness:* any solution found by the procedure is indeed a solution of the verification technique i.e. the solution is correct.
2. *Completeness:* whenever there is a solution of the verification technique, there should exist a path that leads to the solution i.e. no possible solution is left out.
3. *Termination:* there is no infinite branch. This is not guaranteed when dealing with unbounded cases.

Protocol verification is not an easy task to perform manually; therefore tools have been developed to automatically check whether a protocol can be attacked. The biggest challenge to automated tools is undecidability caused by unbounded number of sessions and this could make even simple properties like secrecy to be undecidable [30], hence many techniques focus on bounded cases. Examples of tools that focus on bounded cases are Avispa [5] and Scyther [18].

What is ProVerif?

Tools that handle unbounded cases have also been developed. One such tool is ProVerif which is considered as the most mature in this approach [17]. ProVerif is an automatic cryptographic protocol verifier for the Dolev-Yao attacker model; Figure 3.8 shows its architecture. ProVerif takes protocols written in a variant of the applied pi calculus called *typed pi calculus* as input together with some security property to be verified. The protocol is then automatically translated into a set of first-order Horn clauses [46, 11] and the properties are translated into derivability queries. The resolution can have three outcomes: the property is proven and true, or the analysis does not terminate or the property cannot be proved (which means it is false), at which point ProVerif tries to reconstruct an attack to help the designer see how the property could be broken by an attacker, however, there is a possibility that ProVerif may not be able to reconstruct the attack.

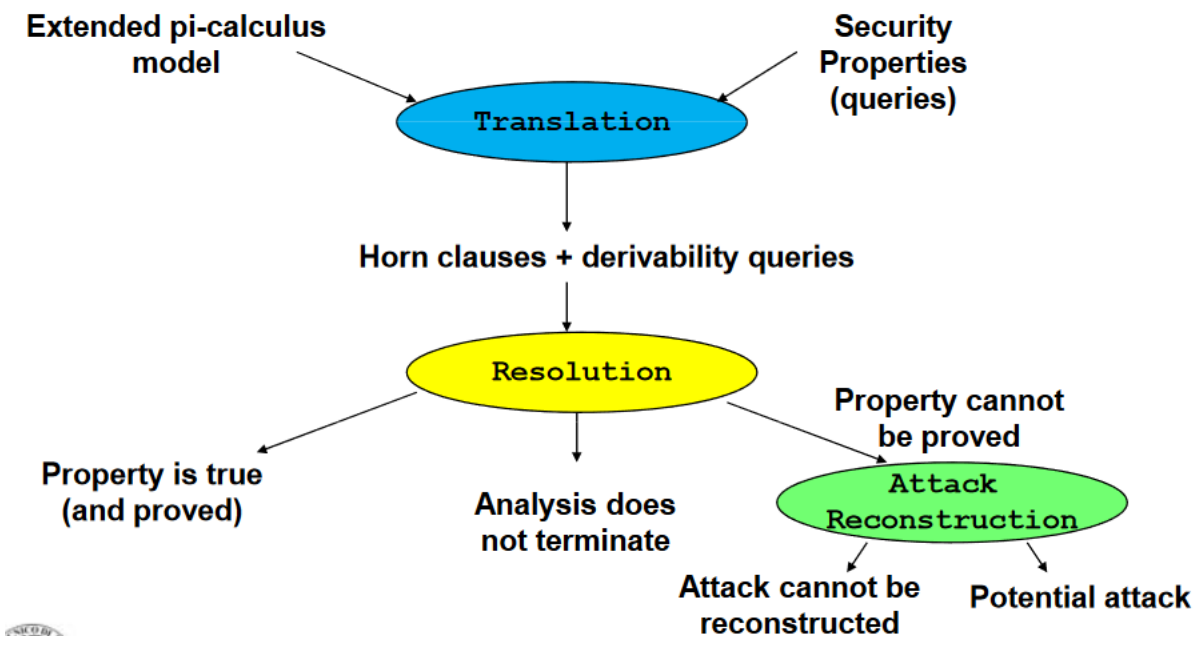

Extended pi-calculus model
Security Properties (queries)
Translation
Horn clauses + derivability queries
Resolution
Property is true (and proved)
Analysis does not terminate
Property cannot be proved
Attack Reconstruction
Attack cannot be reconstructed
Potential attack

Figure 3.8: ProVerif Architecture[39]

ProVerif is capable of proving the following properties.

- Secrecy
- Authentication (and correspondence assertions)
- Strong secrecy
- Equivalence properties between processes that differ only by terms.

ProVerif Possibilities and Limitations [12]

- Protocol analysis is considered with respect to an unbounded number of sessions and an unbounded message space owed to some well-chosen approximations.
- Because of the use of approximations, ProVerif can give false attacks, but if it claims that the protocol satisfies some property, then the property is actually satisfied.
- It is capable of attack reconstruction: when a property cannot be proved, ProVerif tries to reconstruct an execution trace that falsifies the desired property.

We now present how to model a protocol using the typed pi calculus. Consider the simplified Request Signature protocol's first version:

Basics of Typed pi Calculus

A —> B:m
B —> A:sign(m,sskB)

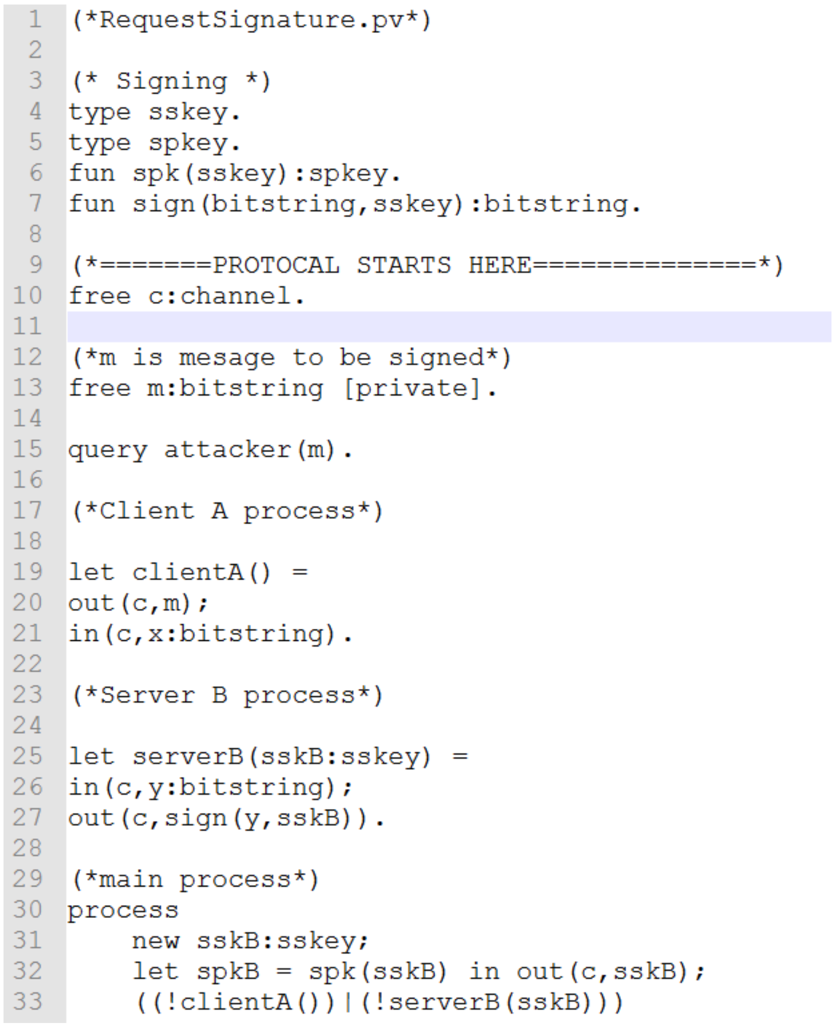
We want to model this protocol in ProVerif and test that m is secret. Figure 3.9 provides the typed pi calculus model for the above protocol description.
The typed pi calculus provides a type for every variable or name but largely resembles the applied pi calculus in its syntax. ProVerif uses the typed pi calculus to model protocols.

Source Structure

ProVerif source consists of:
- a sequence of declarations
- the word "process" followed by statements describing the steps of the process.
  $\langle decl \rangle^{*} process \langle process \rangle$
- decl := free names, constructors, destructors, queries, process macros

ProVerif protocol source files are simply text files saved with a *.pv* extension (line 1). Comments are included in the source file by using *(**)*. The keyword *type* is used to declare new types (line 4,5). Every statement outside a process macro ends with a period *(.)*. Constructors are declared using the keyword *fun* (line 6, and 7). For each constructor that returns a term, the type of the term must be indicated, e.g. line 6, and 7. All names or variables must have types, e.g. line 10, and 13. The key word *free* is used to declare global names i.e. those that are accessible to all process including an attacker (line 10). For names that must be hidden[1] from the attacker's knowledge, the keyword *[private]* is appended at the end of each declaration (line 13). All queries that check whether a particular property holds for the protocol are declared using the keyword *query* e.g. line 15. Secrecy queries are checked using the function *attacker(...)*, with the name to be checked as the parameter, while correspondence assertions are declared using events (discussed in later sections).
Process macros are declared to avoid writing all the steps for each process in the main process (line 30-33). Hence the client process is declared at lines 17-21 and the

[1]Such names are only considered to be initially hidden from the attacker unless proven to be truly secret.



```
 1 (*RequestSignature.pv*)
 2
 3 (* Signing *)
 4 type sskey.
 5 type spkey.
 6 fun spk(sskey):spkey.
 7 fun sign(bitstring,sskey):bitstring.
 8
 9 (*=======PROTOCAL STARTS HERE==============*)
10 free c:channel.
11
12 (*m is mesage to be signed*)
13 free m:bitstring [private].
14
15 query attacker(m).
16
17 (*Client A process*)
18
19 let clientA() =
20 out(c,m);
21 in(c,x:bitstring).
22
23 (*Server B process*)
24
25 let serverB(sskB:sskey) =
26 in(c,y:bitstring);
27 out(c,sign(y,sskB)).
28
29 (*main process*)
30 process
31     new sskB:sskey;
32     let spkB = spk(sskB) in out(c,sskB);
33     ((!clientA())|(!serverB(sskB)))
```

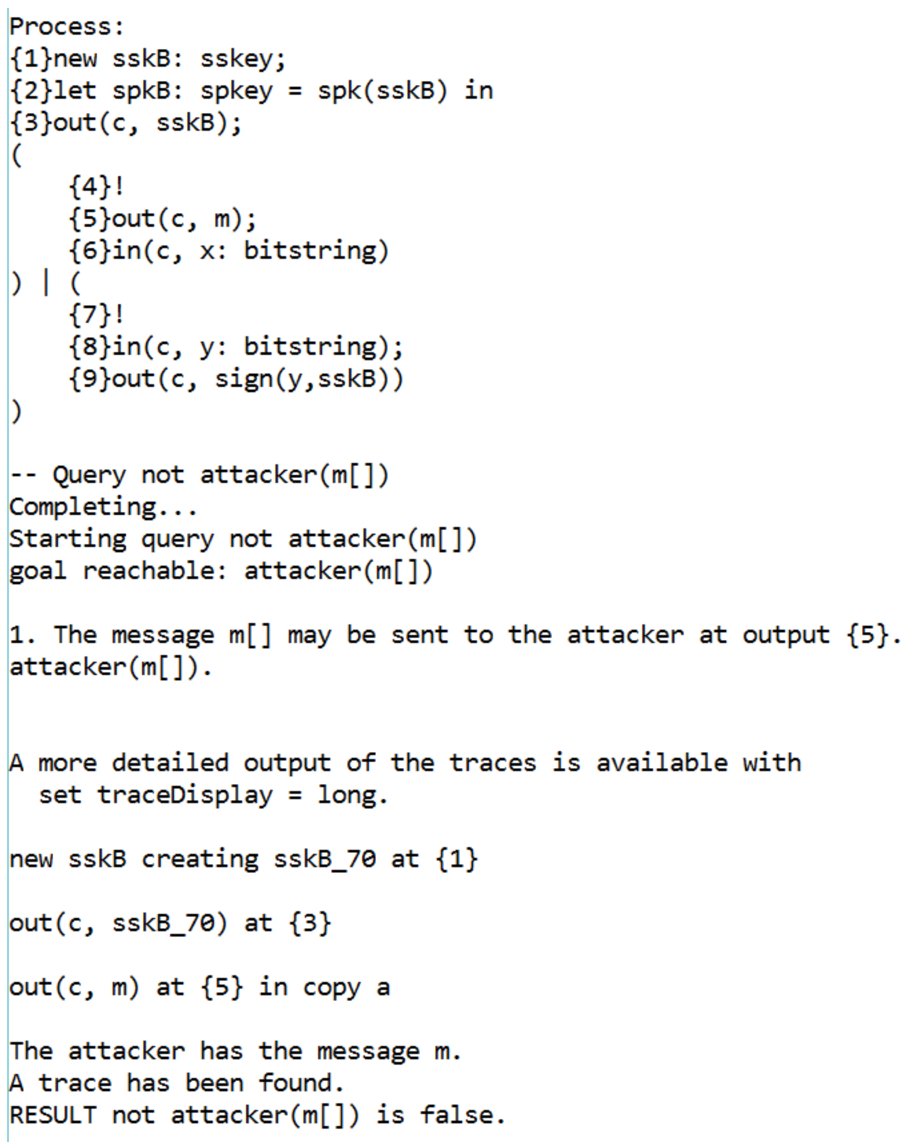
Figure 3.9: Typed pi Calculus for Request Signature

server process at lines 23-37. Each statement in a process macro ends with a semi colon and the last statement ends with a period to signify the end of the process.
At line 10, we declare a public channel c for the client and server to use for communication. At line 13 we declare the secret message m. At line 15 we query the secrecy of m. Internally, ProVerif attempts to prove that a state in which the name m is known by the attacker is unreachable, hence it tries to prove that **not** *attacker(m)* is true. If the output says *not attacker(m) is false*, it means that an attacker can discover the secret m.
The client process A basically follows the two steps of the protocol described above. First it sends the message m to be signed, using the public channel c (line 20) and afterwards begins to wait for the signed message which would be saved in the variable x (line 21).
The server process B similarly waits for an incoming message on the public channel c to be saved in variable y (line 26) and then signs it using its private signing key (*sskB*) and sends it back on the channel (line 27).
The main process first creates a new private signing key (also known as secret signing key—*sskB*) at line 31. It then gets the public signing key of *sskB*, saves it as *spkB*

and sends it out on the public channel for anyone to be able to verify the signature at any point[2] (line 32). Line 33 instantiates the two processes and runs them in parallel.

ProVerif Output

The output of line 15—*query attacker(m)* is given in figure 3.10:



```
Process:
{1}new sskB: sskey;
{2}let spkB: spkey = spk(sskB) in
{3}out(c, sskB);
(
    {4}!
    {5}out(c, m);
    {6}in(c, x: bitstring)
) | (
    {7}!
    {8}in(c, y: bitstring);
    {9}out(c, sign(y,sskB))
)

-- Query not attacker(m[])
Completing...
Starting query not attacker(m[])
goal reachable: attacker(m[])

1. The message m[] may be sent to the attacker at output {5}.
attacker(m[]).


A more detailed output of the traces is available with
  set traceDisplay = long.

new sskB creating sskB_70 at {1}

out(c, sskB_70) at {3}

out(c, m) at {5} in copy a

The attacker has the message m.
A trace has been found.
RESULT not attacker(m[]) is false.
```

Figure 3.10: ProVerif Output for Request Signature

ProVerif begins by outputting the steps of each process in the model and then begins to verify each query, in this case we only have one query checking the secrecy of m. The final result is *“RESULT not attacker(m[]) is false.”* which means that the secret m is reachable by the attacker. ProVerif first states which query it is verifying using the statement *“– Query not attacker(m[])”*. It then provides an En-

[2]The destructor for verifying signed messages is not included in this example but is presented in later sections

glish description of the *derivation* denoted by *“1. The message m[] may be sent to the attacker at output 5.”*; A derivation is ProVerif’s internal representation of how an attacker may break the property being tested. After the English description, ProVerif then provides a trace of actual steps to take in the protocol in order to break the property. The English description and trace are only given if the goal is reachable by an attacker.

From the output, it is then left with the Designer to fix the protocol in order to preserve the desired security properties. This is the work presented in the Architectural Evaluation section to formally verify that the proposed architecture is sound with regard to secrecy and authentication aspects.

In summary, verification of security protocols is conducted on abstract models of the protocols. Protocols are modeled in the applied pi calculus using cryptographic primitives. The verification is performed by an automated tool—ProVerif (as is the case in this thesis), while taking into account the attacker model (Dolev-Yao in this instance). In the next chapter, we present the specification of the proposed architecture.

4
Specification

This chapter presents a description of the domain, domain model, requirements and use cases. The domain description provides a list of the key entities of the domain and their roles. The domain model is a diagram expressing relations between the constructs in the domain. Requirements present the goals to be achieved to fulfill the research objectives and use cases are later presented to describe the direction taken to achieve the goals.

4.1 Domain Description

The proposed architecture consists of five main entities: a Surveillance Organisation (SO), a Data Protection Authority (DPA), a Court, a Time Stamping Authority (TSA) and a Citizen—also known as Data Subject (DS). This section, together with Figure 4.1, presents the concepts related to each of the above entities in the domain and how they are related.

4.1.1 Surveillance Organisation (SO)

The SO stores surveillance records about citizens. Each surveillance record is called an *Observation.* For each observation, the SO must obtain a *TimeStamp* from the TSA and a *Receipt* from the DPA; this is mandatory if such an observation should become a *Record* that could be used as *Evidence* in Court.

4.1.2 Data Protection Authority (DPA)

The purpose of the DPA is to bring about transparency in the operations of the SO by receiving commitments from the SO about its surveillance observations. The DPA issues receipts for each timestamped observation committed to it. This enables the DPA to respond to citizens who may wish to find out if they had been under surveillance before, thus fulfilling the "Individual Participation Principle" proposed by the OECD.

4.1.3 Court

The Court is the legal entity that has the sole right of making "public" the surveillance records of a citizen. The court issues a court *Order* to the SO for surveillance

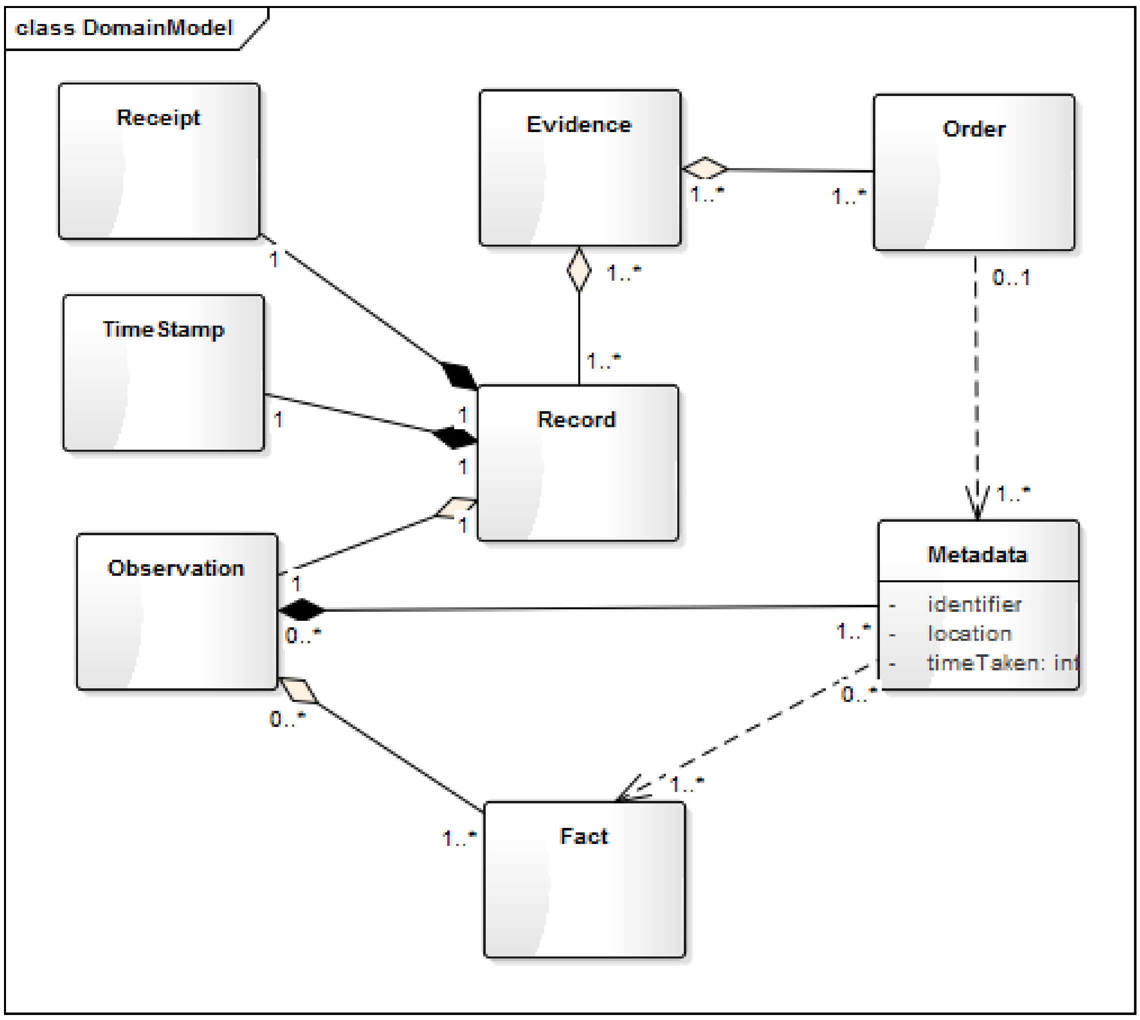


Figure 4.1: Domain Model

records on a particular citizen. Each order may refer to particular *metadata* such as an *identifier* for citizen or some *location*. The SO then responds by disclosing surveillance records matching the order, if any. However, these records are only considered valid if and only if they were timestamped by the TSA and signed (receipted) by the DPA. The court must *verify* that the submitted records are indeed valid otherwise are discarded. The combination of a court *Order* and a *Record* composes *Evidence* that can be used in an court case.

4.1.4 Time Stamping Authority (TSA)

The TSA's sole purpose is to timestamp observations sent by the SO. Each *timestamp* must correspond to exactly one observation or set of observations sent in one session.

4.1.5 Citizen

A citizen represents any individual that is a data subject of the SO. A citizen can make requests to the DPA for the purpose of finding out if he/she has been under surveillance. The DPA must then respond with partial sets of information sufficient to answer the request.

4.2 Vocabulary

Fact: A Fact is anything observed about a citizen; this could be raw unstructured data like video surveillance or imagery.

Metadata: Metadata is structured information extracted from facts; this could be identifiers of citizens (e.g. biometric code, social security number etc), location and time of the fact.

Observation: An Observation is an identifiable record of surveillance that has potential to be used against a citizen in a court of law. An observation has one or more Metadata records and may also be linked to some fact. A fact could exist without being used as an observation; however, metadata only exists to serve the purpose of constructing an observation.

Record: A Record is a timestamped, and receipted observation. This is the only kind of observation that is allowed to be sent to a Court upon a court a order.

TimeStamp: A timestamp relates to the time an observation was taken; in this context it relates to the time an observation is stamped by the TSA in readiness for registration with the DPA.

Receipt: A signature from the DPA confirming receipt of a commitment from the SO.

Order: A request for Records matching particular Metadata e.g. location, or identifiers

Evidence: Verified Records matched with an Order constitute Evidence.

4.3 Requirements

Table 4.2 describes the requirements for the architecture presented in this thesis. Note that this list of requirements focuses on security and other qualities like availability or performance etc are not addressed. The first column provides the functional requirements. The second one provides the quality requirements associated with each functional requirement presented in the first column. The third column presents the security goal that was intended to be achieved by each requirement in the corresponding row.

There are three main functional requirements (1) The SO shall register observations with the DPA, (2) The SO shall disclose records to the Court following a valid court order and (3) A Citizen shall be able to ask the DPA about any surveillance records related to him. For each of these requirements, a number of quality requirements are included which ensure the preservation of the security properties desired in each case. For instance, to ensure that observations sent to the DPA can be trusted regarding when they actually took place, we require that the SO first timestamps them and then commits, furthermore, we ensure that timestamps are unique for each session so that the SO does not reuse timestamps on observations taken at different times. This procedure is followed for the rest of the requirements and the reader is referred to Table 4.2 for more details.

Table 4.1 describes which entities can access which items in the domain. This shows only the items that must be secret to some entities. ✓means should see, X means should not see and 0 means unimportant. *Identifier* is what may be used by a Citizen to request the DPA for records. The DPA only knows identifiers sent by citizens but not in the commitments.

Table 4.1: Secrecy Table

	SO	TSA	DPA	Court	Citizen
Observation	✓	X	X	✓	X
Timestamp	✓	✓	✓	✓	✓
Identifier	0	0	✓	0	✓

4.4 Use Cases

Figure 4.2 presents three business usecases: *create record, create evidence* and *request records.* The create record use case is initiated by the SO to get an *observation timestamped* by the TSA and *signed* by the DPA. The create evidence usecase is initiated by the Court to issue an order for records to the SO and receive the requested records if available. The request records use case is initiated by the Citizen to make an inquiry with the DPA regarding any surveillance records relating to the Citizen.

4.4.1 Brief Use Cases

The following are brief descriptions of the use cases presented in figure 4.2. Each description outlines the steps taken by the actors to achieve the goal of the use case. Note that these are not detailed usecases hence only main scenarios are described while alternative scenarios are omitted. In place of alternative scenarios, mis usecases are presented in Section 4.4.2.

4.4.1.1 UC1: Create Record

Goal: To timestamp and receipt an observation
Actors: SO, TSA, DPA
Main Scenario:

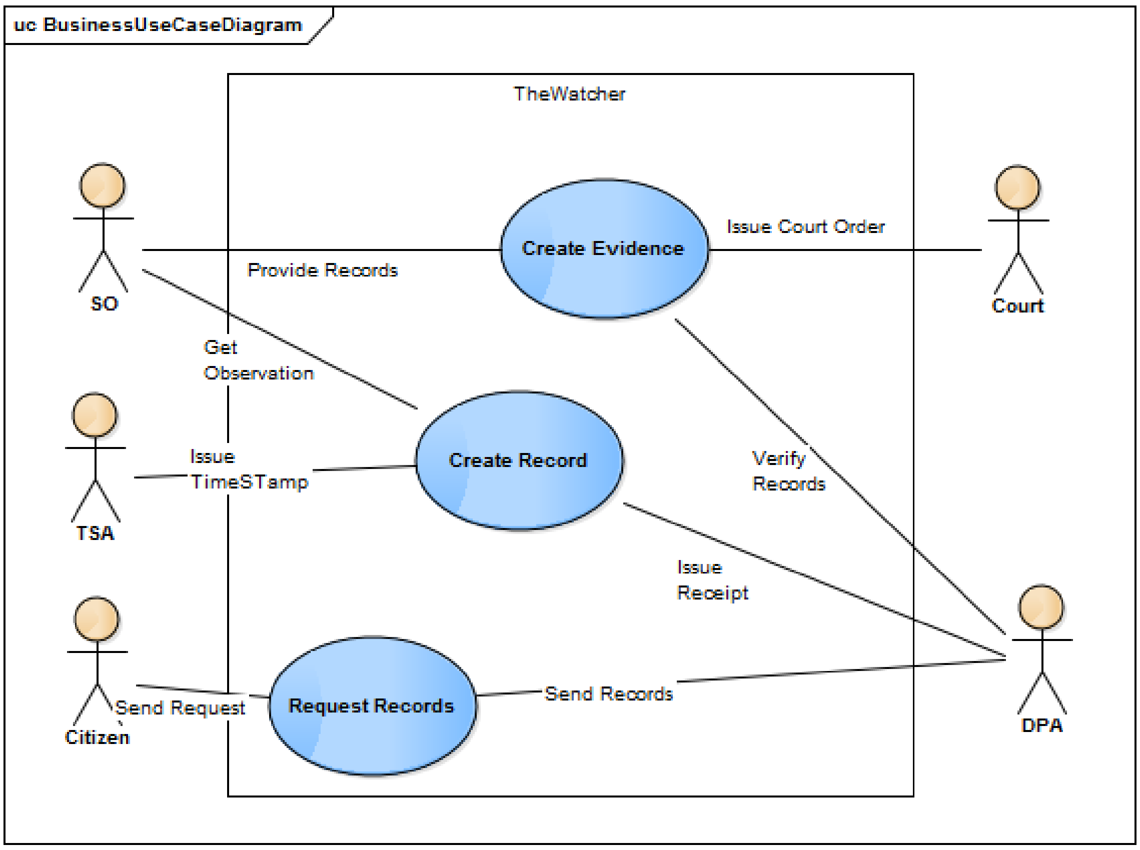


Figure 4.2: Business Use Case Diagram

1. SO sends observation to TSA for timestamping.
2. TSA issues timestamp
3. TSA sends timestamped observation to SO
4. SO sends time stamped observation to DPA for receipt
5. DPA receipts observation
6. DPA sends receipted observation, a.k.a Record, to SO
7. SO saves Record

4.4.1.2 UC2: Create Evidence

Goal: To collect and verify surveillance records for a citizen
Actors: Court, SO, DPA
Main Scenario:

1. Court sends order for records to SO
2. SO sends records matching the given order
3. Court verifies records with DPA
4. If confirmed, Court creates evidence from both the order and the records, else records are discarded

4.4.1.3 UC3: Request Records

Goal: To know of any surveillance operations performed on the citizen
Actors: Citizen, DPA
Main Scenario:

1. Citizen sends request to DPA
2. DPA checks among commitments for those matching Citizen
3. DPA notifies Citizen

4.4.2 Misuse Cases

This section presents the actions that should not be allowed in the system; these are also considered as attacks to the system. As stated earlier, misuse cases have goals conflicting with the regular use cases. We do not provide much detail about each misuse case as they are self explanatory. Table 4.3 presents the misuse cases. In the first column is the actual misuse case, in the second is the security attribute breached by the misuse case and the third column presents the quality requirement from Table 4.2 that prevents the attack).

4.4.3 Requirement–Use Case Matrix

Table 4.4 shows the relationship between use cases and requirements; in particular, for each functional requirement RQ (row), each use case UC (column) that is required to fulfill the requirement is marked with an X.

Table 4.2: Architecture Requirements

Functional Requirement	Quality Requirement	Security Goal
FR1: The SO shall register observations with the DPA	QR1a: Observations shall be time stamped by TSA.	Integrity
	QR1b: Timestamps shall be unique for each observation i.e. no timestamp shall be used for more than one observation.	
	QR1c: The TSA shall sign the timestamped observation to ensure authenticity of timestamps.	
	QR1d: The DPA shall sign only timestamped observations.	
	QR1f: The DPA shall check that timestamps are not older than a period of time predefined by the DPA itself.	
	QR1e: Observations shall remain secret while being sent to either the TSA or the DPA.	Secrecy
FR2: The SO shall disclose surveillance records to a Court upon receipt of a valid Court Order	QR2a: Records shall be disclosed ONLY following a corresponding court order	Secrecy (Confidentiality)
	QR2b: The Court shall check the consistency of the records supplied by the SO with what was registered with the DPA.	Integrity
	QR2c: The SO shall disclose only records committed prior to the court order.	
FR3: A Citizen shall be able to request records of surveillance relating to him	QR3a: The DPA shall access only a partial set of information (see Section 6.2.2) from observations, to allow it to service requests from Citizens.	Secrecy
	QR3b: The DPA shall service citizen requests without accessing secret observations.	
For all FRs above	QR4: All entities shall provide proof of identity during their communication.	Authentication

Table 4.3: Misuse Cases

Mis Use case	Security Attribute Breached	QR to prevent attack
SO commits untimestamped observation	Integrity	QR1a, QR1d
SO commits observation with old timestamp i.e. SO timestamps observation but does not commit immediately	Integrity	QR1f
SO forges timestamp	Integrity	QR1c
SO discloses observations without court order	Secrecy	QR2a
SO discloses uncommitted observations	Integrity	QR2b
SO commits observation after receiving court order for it	Integrity	QR2c
SO uses same timestamp for more than one session (or more than one observation)	Integrity	QR1b
DPA access secret observations	Secrecy	QR1e, QR3b
DPA sends secret observations to Citizen	Secrecy	QR3a
TSA access secret observation	Secrecy	QR1e
Any of the agents sends data without authenticating	Authentication, Secrecy	QR4

Table 4.4: Requirement–Use Case Matrix

	UC1: Create Record	UC2: Create Evidence	UC3: Request Records
RQ1: SO registers observation with DPA	X		
RQ2: SO discloses records to Court	X	X	
RQ3: Citizen requests DPA for records			X

5 Design

This chapter presents the proposed architecture's context diagram, component diagram and sequence diagram.

5.1 Context Diagram

Figure 5.1 presents the contex diagram of the architecture.

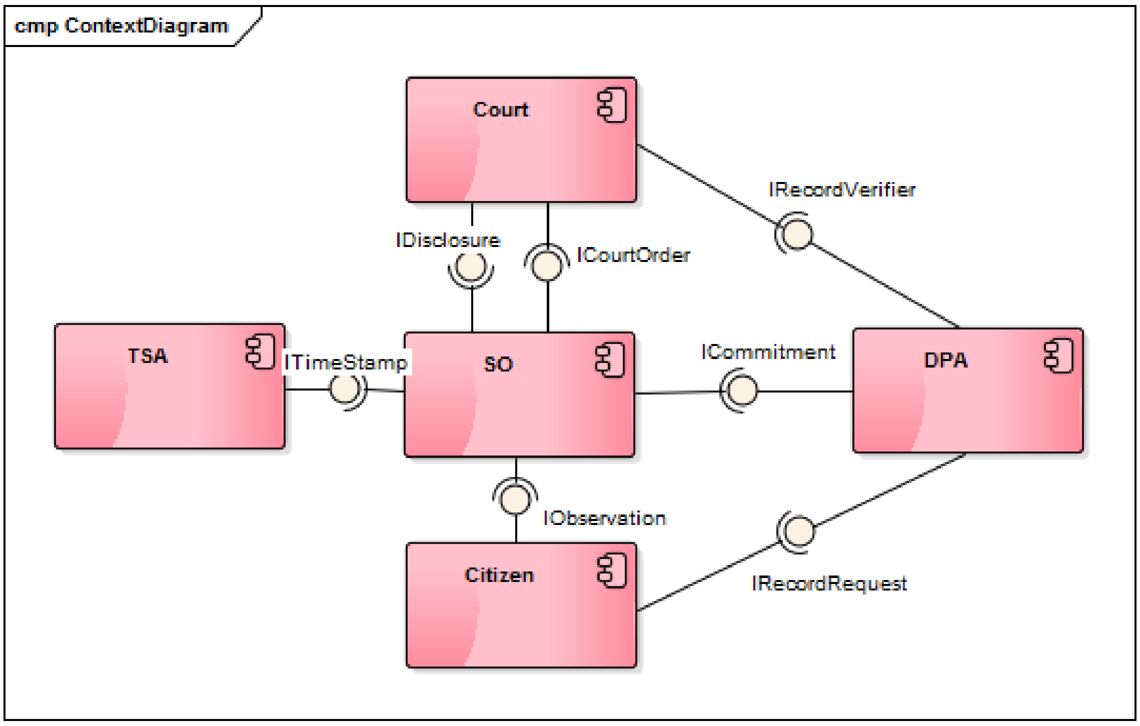


Figure 5.1: Context Diagram

The DPA provides the *ICommitment* interface to the SO for the latter to be able to submit commitments of its observations. The TSA provides the *ITimeStamp* interface for the SO to be able to obtain timestamps for its observations. The SO provides the *ICourtOrder* interface for the Court to be able to issue orders for records to the SO, while the Court provides the *IDisclosure* interface for the SO to be able to respond to orders. The DPA provides the *IRecordVerifier* interface for the Court to be able to verify records it receives from the SO before saving them as evidence. Also, the DPA provides the *IRecordRequest* to the Citizen for the latter to request for any surveillance records related to him. The *IObservation* is for the SO to perform surveillance operations on the Citizen and is included here merely for clarity purposes but is not part of the proposed solution as it only concerns the SO.

5.2 Component Diagram

Figure 5.2 presents the component diagram of the architecture.

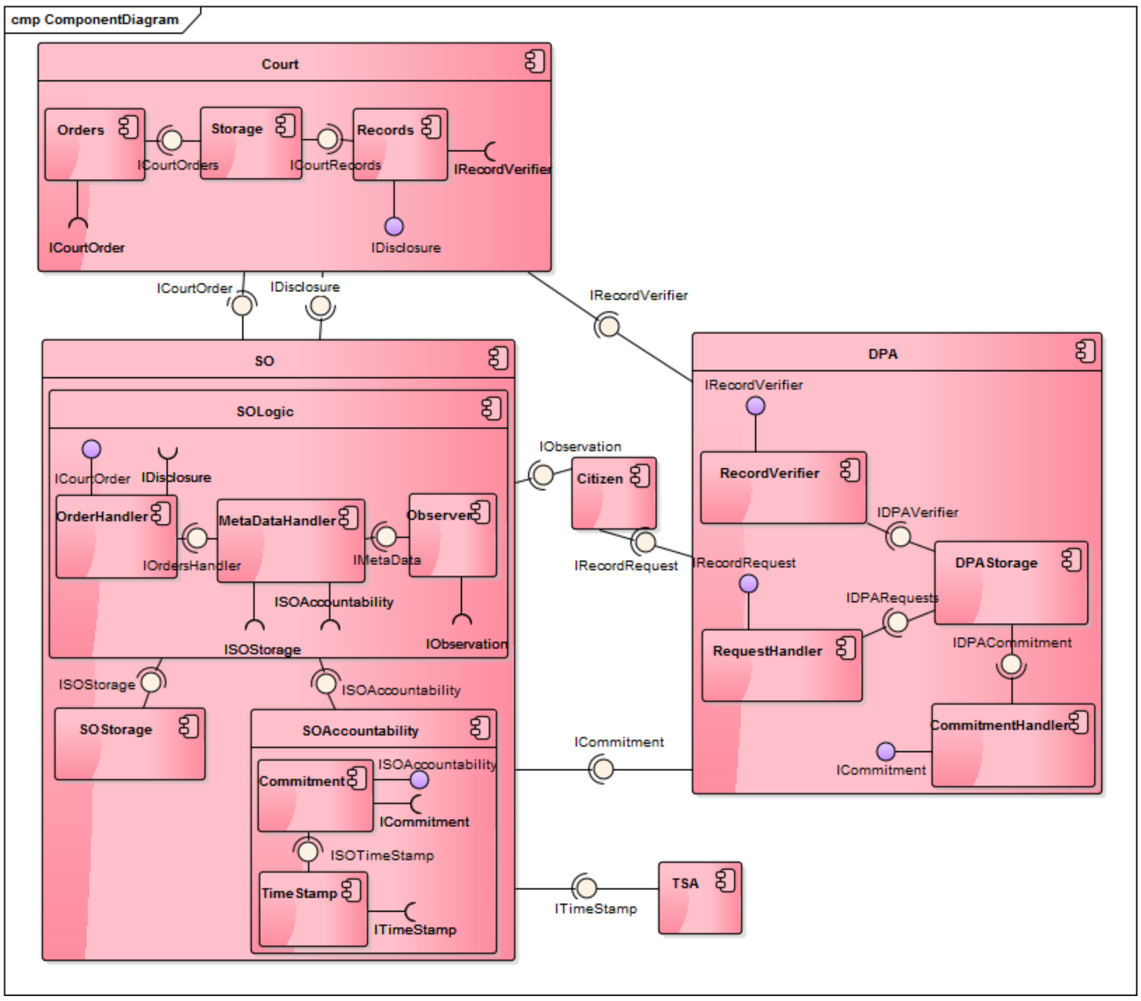


Figure 5.2: Component Diagram

The component diagram presented in figure 5.2 concentrates on the components relevant to the thesis problem. The reader will notice that since the thesis does not focus on qualities such as performance and availability of the entire system, components to handle such issues are not included; for instance the *Storage* components could be structured in different ways to use patterns such as the Replicated Component Group [24] for better availability, but as stated, we only present it here as one functional component to demonstrate our solution. Furthermore, detailed security architectural patterns that include components e.g. system logs, audit interceptors etc are not shown here. Therefore this component diagram focuses on how the components proposed in this thesis relate to each other. The following is an explanation of the SO, DPA and Court components together with their sub-components and interfaces; the TSA and Citizen components are not explained as nothing about them has changed from the context diagram.

5.2.1 SO Component

At the top level, the SO has three components prefixed with SO; the *Logic*, *Accountability* and *Storage* components.

SOLogic: The SOLogic component consists of the *OrderHandler*, *MetaDataHandler* and *Observer* components. The *Observer* is the main surveillance component that gathers facts from citizens. The *OrderHandler* processes orders issued by the Court. It receives orders through the *ICourtOrder* interface and then forwards them to the *MetaDataHandler* through the *IOrdersHandler* interface provided by the latter. The *MetaDataHandler* collects metadata from the *Observer* through the *IMetaData* interface, which it saves in the *SOStorage* component using the *ISOStorage* interface. Upon request from the Court, the *MetaDataHandler* searches for records matching the order and forwards the records to the *OrderHandler* component which in turn discloses the records through the *IDisclosure* interface.

SOStorage: This component handles data storage for the SO

SOAccountablity: The SOAccountability component consists of the *Commitment* and *TimeStamp* components. In order to perform a commitment of an observation to the DPA, the observation would be passed from the *MetaDataHandler* to the *Commitment* component through the *ISOAccountability* interface. The *Commitment* component passes the observation to the SO *TimeStamp* component through the *ISOTimeStamp* interface, which in turn sends the observation to the TSA for timestampiing through the *ITimeStamp* interface. The TSA responds to the SO *TimeStamp* component which then sends the timestamped observation to the *Commitment* component. The *Commitment* component then sends the observation to the DPA through the *ICommitment* interface.

5.2.2 DPA Component

The DPA component consists of a *RecordVerifier, RequestHandler, CommitmentHandler* and *DPAStorage* component.

RecordVerifier: The *RecordVerifier* handles verification of records submitted to the Court by the SO. This is done through the *IRecordVerifier* interface it provides to the Court. The *RecorfVerifier* checks the records submitted by the Court against its internal storage of commitments using the *IDPAVerifier* interface.

RequestHandler: The *RequestHandler* processes requests from citizens sent through the *IRecordRequest* interface. Upon receipt of a request the *RecordHandler* searches for any matching records of the citizen from the *DPAStorage* using the *IDPARequests* interface.

CommitmentHandler: The *CommitmentHandler* processes commitments from the SO.

DPAStorage: Depicts all storage for the DPA.

5.2.3 Court Component

The Court component consists of an *Orders, Records,* and *Storage* component.

Orders: The *Orders* component allows the Court to issue orders to the SO through the *ICourtOrder* interface.

Records: The *Records* component handles receipt of records from the SO through the *IDisclosure* interface, and verifies the records through the *IRecordVerifier* interface provided by the DPA. Once verified, the records are saved to the *Storage* component using the *ICourtRecords* interface.

Storage: Depicts database storage for the Court

5.3 Sequence Diagrams

Figure 5.3 is a presentation of a sequnce diagram that describes the order of events for the communication between all the entities excluding the Citizen i.e. SO, TSA, DPA and Court. We combine the steps of UC1—Create Record and UC2—Create Evidence, to ensure that all steps are carried out in the desired order. This sequence diagram shall be formalised into a protocol to be verified.

The following are the steps depicted in the sequence diagram in figure 5.3:

1. SO sends observation to TSA for time stamping.
2. TSA issues timestamp
3. TSA sends time stamped observation to SO
4. SO sends time stamped observation to DPA for receipt
5. DPA receipts observation
6. DPA sends receipted observation, a.k.a Record, to SO
7. SO saves Record
8. Court sends order for records to SO
9. SO sends records matching the given order
10. Court verifies records with DPA
11. If confirmed, Court creates evidence from both the order and the records, else records are discarded

Similarly, the sequence diagram for a citizen request to the DPA is depicted in Figure 5.4. The citizen first sends his identity to the DPA. The DPA uses identity to check for any matching commitments from the SO. The DPA then responds with a message indicating whether records have been found or not.

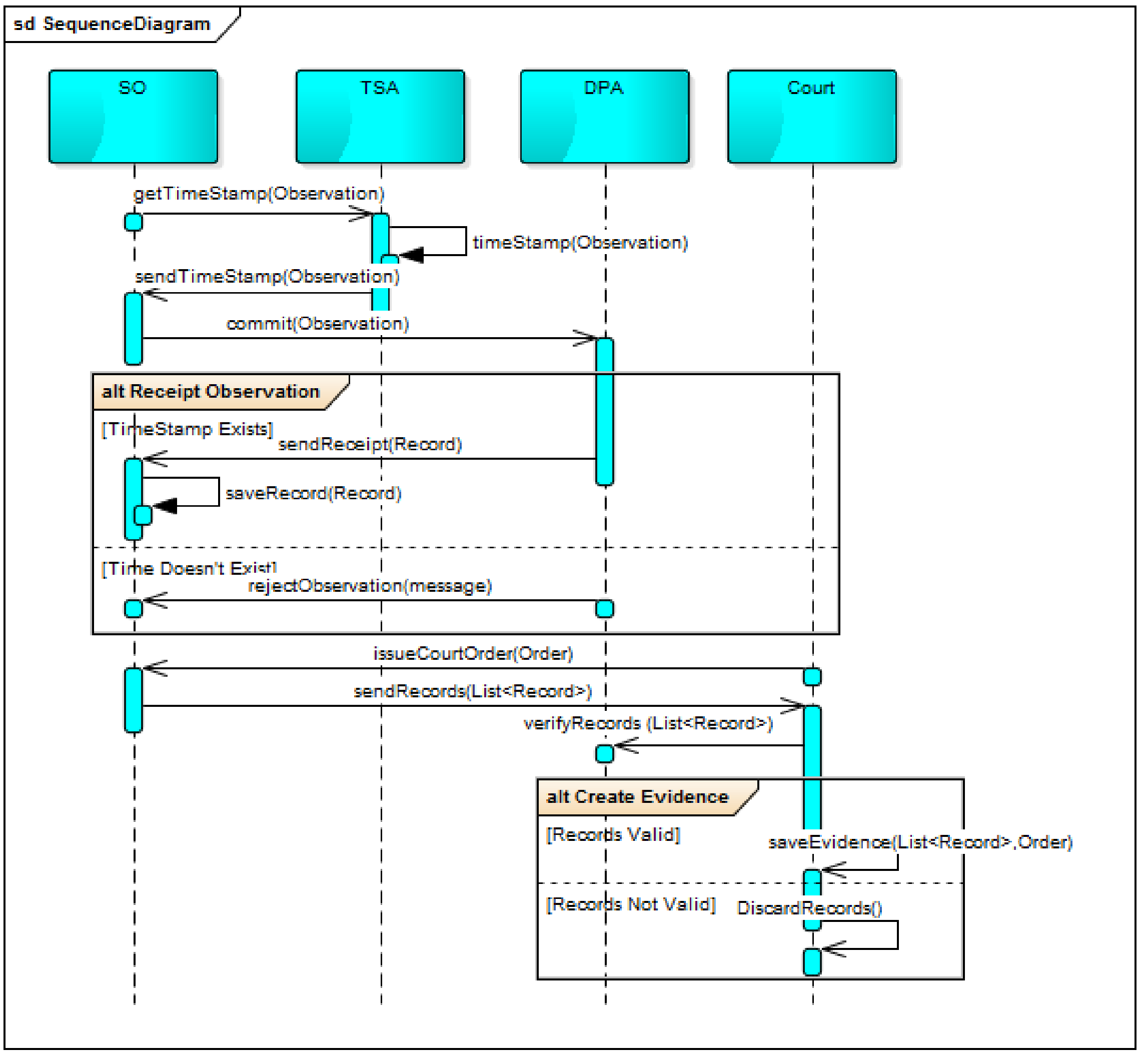


Figure 5.3: Create Record and Evidence Sequence Diagram

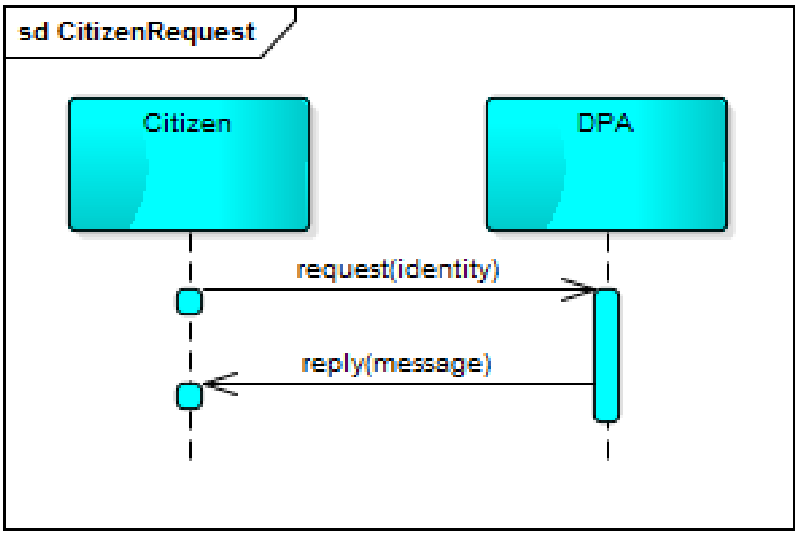


Figure 5.4: Citizen Request Sequence Diagram

6
Evaluation

To evaluate the proposed architecture with regard to its fulfillment of the security properties expressed in the requirements (Table 4.2), the sequence diagrams of the architecture are modeled as security protocols in the applied pi calculus and then formally verified using ProVerif. Particularly, the architecture must preserve *secrecy* of the surveillance observations, and *authentication* of participating agents. Furthermore, *correspondence assertions* are also used to ensure that all agents perform events in the required order.

6.1 Protocol Description

We first start by providing informal narrations of each protocol using sequences of messages.

Create Record and Evidence

We refer to the combined actions of UC1 and UC2 presented in figure 5.3. The following are the actions:

1. SO sends observation to TSA for time stamping.
2. TSA issues timestamp
3. TSA sends time stamped observation to SO
4. SO sends time stamped observation to DPA for receipt
5. DPA receipts observation
6. DPA sends receipted observation, a.k.a Record, to SO
7. SO saves Record
8. Court sends order for records to SO
9. SO sends records matching the given order
10. Court verifies records
11. If confirmed, Court creates evidence from both the order and the records, else records are discarded

We start by writing an informal description of the entire protocol, then model it in a step-wise fashion: by adding agents to, and security properties to preserve in the protocol, in an incremental manner as explained in Section 1.3.2.2. For instance, we start by considering the communication between the SO and the TSA, and the security property *secrecy* and then continue with this same set of agents to also test for *authentication* using *correspondence events*. Later we add the communication between the SO and the DPA for each of the security properties stated earlier, the Court and the SO, and conclude with the communication the Citizen and the DPA.

We shall now write an informal description of the protocol as follows:
Let s be an observation, t be a timestamp, f be a signature, o be a court order, and CT be the Court.

1. *SO —> TSA: s*
2. *TSA —> SO: (s,t)*
3. *SO —> DPA: (s,t)*
4. *DPA —> SO: ((s,t),f)*
5. *CT —> SO: o*
6. *SO —> CT: (o,s)*
7. *CT —> DPA: s*
8. *DPA —> CT:s*

Note that the last two steps where CT verifies an observation s by sending it to the DPA may not be necessary since CT may use a different technique e.g. public signing key of the DPA to check the signature of the DPA on s thus making the verification internal to the Court and reducing the steps to:

1. *SO —> TSA: s*
2. *TSA —> SO: (s,t)*
3. *SO —> DPA: (s,t)*
4. *DPA —> SO: ((s,t),f)*
5. *CT —> SO: o*
6. *SO —> CT: (o,s)*

Request Records
The narrations for a citizen request sent to the DPA are as follows: let *CIT* be a Citizen, i be an identifier of the citizen and r be the response from the DPA.

1. *CIT —> DPA: i*
2. *DPA —> CIT: r*

6.2 Incremental Modeling and Verification

This section presents a model and verification of the entire protocol which includes all the use cases. First we start by showing a step by step verification for the communication between the SO and the TSA for the *secrecy* property so that the reader can gain an understanding of why we take which steps in the protocol. We present six versions of this partial model and for each version we explain what problems or attacks on secrecy could be encountered.

6.2.1 Secrecy

This section presents models related to secrecy and considers communications between the SO and TSA only.

6.2.1.1 Version 1.0: All Plain

This first version sends plaintext messages between agents. Lines 1-26 are declarations, while the actual protocol steps are from line 30 to 47. This is in fulfillment of quality requirement QR1a which requires observations to be timestamped.

Symmetric key encryption

```
2        type key.
3        fun senc(bitstring,key):bitstring.
4        reduc forall m:bitstring,k:key; sdec(senc(m,k),k) =
   ↪ m.
```

Lines 2 to 4 declare symmetric encryption; line 2 declares a type *key* to represent a symmetric key, line 3 declares the binary *constructor senc* which encrypts a given text represented by the built-in type *bitstring*, with the given symmetric *key*. Line 4 declares a *destructor sdec* for the constructor *senc*, which is used to manipulate the terms formed by the constructor, in this instance, to return the plaintext of the encrypted text.

Asymmetric key encryption

```
7        type skey.
8        type pkey.
9        fun pk(skey):pkey.
10       fun aenc(bitstring,pkey):bitstring.
11
12       reduc forall m:bitstring, k:skey; adec(aenc(m,pk(k))
   ↪ ,k) = m.
```

Similarly, lines 7 to 12 declare asymmetric encryption. First, two types are declared– a secret key *skey* and its associated public key *pkey*. Next is declared the unitary constructor *pk(..)* which returns the public key associated with the secret key given as a parameter. Lastly the constructor and destructor for asymmetric encryption are declared as *aenc* and *adenc* respectively.

Digital Signatures

```
15       type sskey.
16       type spkey.
17       fun spk(sskey):spkey.
18       fun sign(bitstring,sskey):bitstring.
```

```
19        reduc forall m: bitstring ,k:sskey; getmess(sign(m,k
   ↪ )) = m.
20
21        (*checksign returns m only if k matches pk(k)*)
22        reduc forall m:bitstring ,k:sskey; checksign(sign(m,k
   ↪ ),spk(k)) = m.
```

Lines 15 to 22 declare functions related to digital signatures which also use public key encryption like asymmetric encryption hence only explain the destructors declared in lines 19 and 22. The destructor *getmess* allows an agent to retrieve a signed message while the *checksign* allows an agent to retrieve a message only if the supplied public key matches the secret signing key that the message was initially signed with. Though we do not use *getmess*, it must be included, otherwise the capabilities of the attacker would be unnecessarily limited, which would lead to the protocol missing some attacks. Hence we include it here to show that signed messages can be retrieved by any agent including an attacker, unless encrypted.

Public channel and secret observation

```
24        free c:channel.
25        (*s is an observation*)
26        free s:bitstring [private].
```

Line 24 declares the public channel used for communication. The attacker listens on everything sent on this channel in conformance to the Dolev-Yao[19] model described in section 3.2.6 . Line 26 declares the name *s* which we assume to be a surveillance observation which should be sent to the TSA for timestamping. Recall that names declared *free* are globally accessible to all agents including an attacker, hence we restrict *s* from the attacker's knowledge by appending the optional parameter *[private]* to its declaration.

Secrecy query

```
28        query attacker(s).
```

Line 28 declares the query that checks whether *s* is secret in any run of the the protocol.

SO process macro

```
32        let clientSO() =
33        out(c,s); (*send s to TSA*)
34        in(c,x:bitstring); (*read timestamped s*)
35        0.
```

Lines 32 to 35 declare the process macro for the SO which outlines the steps the agent playing the role of the SO would take. At line 33, the SO sends the observation *s* on the channel *c* and then begins to wait for a response from the TSA (presumably) at line 34. Once the response comes through, the SO process ends by doing nothing (for now) as designated by the zero process (0).

TSA process macro

```
39      let serverTSA() =
40      in (c,y:bitstring);
41      new t:bitstring; (*t is a time stamp which is
   ↪ modeled as a nonce*)
42      out (c,(y,t));
43      out (c,t). (*make timestamp public*)
```

Similarly, lines 37 to 43 declare the process macro for the TSA. The TSA starts by waiting for an input at line 40 which it would receive in the variable *y* on channel *c*. Upon receipt of the observation, the TSA creates a timestamp which we simply model here as a nonce (represented by a fresh variable of type bitstring) due to the fact that ProVerif can't model timestamps [4]. Timestamp is made publicly accessible by outputting it at line 43. The TSA then sends a pair of the received observation and the newly created timestamp on the channel *c*. This is what the SO would expect to receive at line 34.

Main process

```
45      process
46
47                ((!clientSO())|(!serverTSA()))
```

Lines 45 to 47 describe the main process where all the macros are then run in parallel and with multiple sessions.

We then run the source file to view ProVerif output; the focus is on the query at line 28 above which checks whether *s* is secret for the attacker. Figure 6.1 shows the output of ProVerif for version 1.0 of the model. As expected, the result from ProVerif is: "RESULT not attacker(s[]) is false." which means that it is possible for an attacker to obtain this secret. Looking at the derivation listed at, we notice that the secret is leaked at output 2 (in the ProVerif output) when the SO sends it to the TSA. This is because it is in plaintext and it is being sent on a public channel which is accessible to the adversary. The solution is to use appropriate encryption.

- PROBLEM1a: *s* is leaked at *out(c,(s[,*]))* if not encrypted because *c* is a public channel; [,*] means whether *s* is paired with something else or not, it is vulnerable as long as it's not encrypted (Quality requirement QR1e).
- SOLUTION1a: Use appropriate asymmetric encryption.

```
Process:
(
    {1}!
    {2}out(c, s);
    {3}in(c, x: bitstring)
) | (
    {4}!
    {5}in(c, y: bitstring);
    {6}new t: bitstring;
    {7}out(c, (y,t))
)

-- Query not attacker(s[])
Completing...
Starting query not attacker(s[])
goal reachable: attacker(s[])

1. The message s[] may be sent to the attacker at output
  ↪ {2}.
attacker(s[]).


A more detailed output of the traces is available with
  set traceDisplay = long.

out(c, s) at {2} in copy a

The attacker has the message s.
A trace has been found.
RESULT not attacker(s[]) is false.
```

Figure 6.1: ProVerif Output: Version 1.0

6.2.1.2 Version 1.1: Introduce Asymmetric Encryption

To resolve PROB1a, we let the SO encrypt the secret with the public key of the TSA so that only the latter can decrypt it with its secret key. Version 1.1 implements these changes in fulfillment of quality requirement QR1e that requires secrecy for the observation sent to the TSA. For this version, we only present the relevant parts of the model that will change.

Figure 6.2 shows version 1.1 of the updated model that includes asymmetric encryption. The steps of each process macro still remain the same but asymmetric encryption has been introduced. In this regard, the SO is declared with the public key of the TSA (*pkTSA*) and its own secret key *skSO* as parameters. The *pkTSA* will be useful to encrypt any message being sent to the TSA and the *skSO* will be

```
30      (*SO macro*)
31
32      let clientSO(pkTSA:pkey,skSO:skey) =
33      out(c,aenc(s,pkTSA));
34      in(c,x:bitstring);
35      0.
36
37      (*TSA macro*)
38
39      let serverTSA(pkSO:pkey,skTSA:skey) =
40      in(c,y:bitstring);
41      let sY:bitstring = adec(y,skTSA) in
42      new t:bitstring;
43      out(c,aenc((sY,t),pkSO)).
44
45      (*main process*)
46      process
47              new skSO:skey;
48              new skTSA:skey;
49              let pkSO = pk(skSO) in out(c,pkSO);
50              let pkTSA = pk(skTSA) in out(c,pkTSA);
51              ((!clientSO(pkTSA,skSO))|(!serverTSA(pkSO,
   ↪ skTSA)))
```

Figure 6.2: Model Version 1.1: Asymmetric Encryption

useful for decrypting messages encrypted with the SO's public key.

Line 39 declares the process macro for the TSA with the public key of the SO (*pkSO* and its own secret key *skTSA* as parameters. At line 41 the decrypted message is read into variable *sY* which is then paired with the timestamp *t*, decrypted with SO's public key and then output at line 43. The SO should receive this message at line 41 and decrypt it with its secret key.

In the main process, we create the secret keys for the SO and TSA at lines 47 and 48, we then generate their corresponding public keys and output them on the public channel to make them accessible to the attacker. Line 51 then instantiates each process macro with appropriate parameters.

Figure 6.3 shows the relevant output of ProVerif for the verification of version 1.1 of the model.
The output of ProVerif for version 1.1 states that the secrecy of *s* is preserved. But what is incorrect about this model? QR4 requires that all entities shall require proof of their identity.

- PROBLEM1b–REALITY CHECK: TSA can start a session with anyone; encryption of *(sY,t)* should be based on *pk* of the inter-

```
Process:
...
–– Query not attacker(s[])
Completing...
Starting query not attacker(s[])
RESULT not attacker(s[]) is true.
```

Figure 6.3: ProVerif Output: Version 1.1

locutor.

- SOLUTION1b: Let SO send its *pk* together with *s*, then TSA should also read *pkY* in addition to sY, so that we have *aenc((sY,t),pkY)*

Problem1b is more of a reality check than an attack. In reality, the TSA simply waits for requests for timestamps; it does not matter who the request comes from hence we should not encrypt whatever message it receives with the public key of the SO, rather, it should be encrypted with the public key of the agent who sent the message because it may not always be the case that the messages are from the SO. Version 1.2 allows the SO to send its identity, which for simplicity purposes we assume to be it's public key, together with the secret observation and then allow the TSA to encrypt its response with the public key it receives from the message.

6.2.1.3 Version 1.2: Introduce SO Identity (pkSO) and TSA should accept any interlocutor

Figure 6.4 shows version 1.2 of the model. The difference between version 1.2 and 1.1 is found at lines 33 and 41. At line 33, the SO sends its public key together with the observation *s* while at line 41, the TSA reads both values of the pair into two variables; *sY* for the observation and *pkY* for the public key of the interlocutor. ProVerif still proves that *s* is secret and displays the message "RESULT not attacker(s[]) is true". However, another reality check needs to be addressed:

- PROBLEM1c–REALITY CHECK: *SO* can start a session with any public key including its own. This is also to fulfill QR4 whch requires entities to prove their identities.
- SOLUTION1c: Let SO start by reading a public key from the channel and then use it to start a session with the TSA

6.2.1.4 Version 1.3: SO reads public key of interlocutor from channel

Version 1.3 implements solution 1c by allowing the SO to first read a public key from the channel and then use it to send observations for timestamps. This version is presented in Figure 6.5.

The changes are effected at line 33 where the SO first reads a public key from

```
    (*SO macro*)

    let clientSO(pkTSA:pkey,skSO:skey) =
    out(c,aenc((s,pk(skSO)),pkTSA));
    in(c,x:bitstring);
    0.

    (*TSA macro*)

    let serverTSA(pkSO:pkey,skTSA:skey) =
    in(c,y:bitstring);
    let (sY:bitstring,pkY:pkey) = adec(y,skTSA) in
    new t:bitstring;
    out(c,aenc((sY,t),pkY)).

    (*main process*)
    process
            new skSO:skey;
            new skTSA:skey;
            let pkSO = pk(skSO) in out(c,pkSO);
            let pkTSA = pk(skTSA) in out(c,pkTSA);
            ((!clientSO(pkTSA,skSO))|(!serverTSA(pkSO,
  ↪ skTSA)))
```

Figure 6.4: Model Version 1.2: TSA Accepts any Interlocutor

```
30    (*SO macro*)
31
32    let clientSO(pkTSA:pkey,skSO:skey) =
33    in(c,pkX:pkey);(*SOL1c*)
34    out(c,aenc((s,pk(skSO)),pkX));
35    in(c,x:bitstring);
36    0.
```

Figure 6.5: Model Version 1.3: SO Reads Public key From Channel

the channel into variable *pkX* and then uses it to encrypt its message to the TSA. However, with this new change, ProVerif shows that the secrecy of *s* is no longer preserved. The final result of the output is “RESULT not attacker(s[]) is false”. A careful examination of the derivation reveals that since the SO is ready to send its observations to any public key, it may actually end up sending it to an attacker who will ultimately obtain the secret by decrypting it with his secret key.

- PROBLEM1d: If an attacker has some key k and uses pk(k) to get a public key, he can ultimately get *s* (Quality requirement QR1e).
- SOLUTION1d: Restrict who to send *s* to, by checking that the starting pk is *pkTSA*.

6.2.1.5 Version 1.4: SO only proceeds if supplied pk is that of TSA

Figure 6.6 is a presentation of version 1.4 of the model; it implements solution 1d by checking that the input pk is that of the TSA. This is to still fulfill QR1e requiring secrecy.

```
30    (*SO macro*)
31
32    let clientSO(pkTSA:pkey,skSO:skey) =
33    in(c,pkX:pkey);
34    if pkX = pkTSA then (*SOL1d*)
35    out(c,aenc((s,pk(skSO)),pkX));
36    in(c,x:bitstring);
37    0.
```

Figure 6.6: Model Version 1.4: SO Compares Input pk to pkTSA

Line 34 provides the necessary check suggested in solution 1d. ProVerif proves that the secrecy of *s* is preserved.
The next step is to provide authenticity for the timestamp the SO receives.

- PROBLEM1e: *t* needs to be validated that it truly comes from the TSA (Qaulity requirement QR1c).

- SOLUTION1e: Digitally sign *t* using the signature of the TSA.

6.2.1.6 Version 1.5: Sign *timestamp* with signature of TSA to ensure that it's valid and not forged

Figure 6.7 presents version 1.5 of the model, which partially fulfills quality requirement QR1c by digitally signing the timestamp to provide its authenticity.

```
39      (*TSA macro*)
40
41      let serverTSA(pkSO:pkey,skTSA:skey,sskTSA:sskey) =
42      in(c,y:bitstring);
43      let (sY:bitstring,pkY:pkey) = adec(y,skTSA) in
44      new t:bitstring;
45      out(c,aenc((sY,sign(t,sskTSA)),pkY)).
46
47      (*main process*)
48      process
49              new skSO:skey;
50              new skTSA:skey;
51              new sskTSA:sskey;
52              let pkSO = pk(skSO) in out(c,pkSO);
53              let pkTSA = pk(skTSA) in out(c,pkTSA);
54              let spkTSA = spk(sskTSA) in out(c,spkTSA);
55              ((!clientSO(pkTSA,skSO))|(!serverTSA(pkSO,
   ↪ skTSA,sskTSA)))
```

Figure 6.7: Model Version 1.5: TSA Digitally Signs Timestamp

Line 41 adds the secret signing key *sskTSA* of the TSA to the list of parameters with which the TSA is instantiated. This key is then used to sign the newly created timestamp at line 45 which is sent together with the observation. Note that what is signed is only the timestamp *t*.

- PROBLEM1f: signed *t* could be reused by SO on other observations, yet *t* needs to be fresh and different for each *s* (Quality requirement QR1b).
- SOLUTION1f: Sign pair *(sY,t)* with signature of TSA.

6.2.1.7 Version 1.6: Sign pair of *observation* and *timestamp* with signature of TSA (SOLUTION1f)

Figure 6.8 shows version 1.6 of the model, that allows the TSA to sign the pair of the observation and timestamp rather than the timestamp only thus fulfilling quality requiremnt QR1b which requires timestamps to be unique.

```
39        (*TSA macro*)
40
41        let serverTSA(pkSO:pkey,skTSA:skey,sskTSA:sskey) =
42        in(c,y:bitstring);
43        let (sY:bitstring,pkY:pkey) = adec(y,skTSA) in
44        new t:bitstring;
45        out(c,aenc(sign((sY,t),sskTSA),pkY)). (*SOL1f*)
46
```

Figure 6.8: Model Version 1.6: TSA Signs Pair of Observation and Timestamp

Line 45 shows that the signature is now performed on the pair rather than on the timestamp only.

In summary encryption techniques improve secrecy and digital signatures are used to provide a authenticity or trust for messages received. We use signatures from hence forth whenever we want to ensure trust in a message. In ProVerif there's no need to for message authentication codes as message authentication is implicit in the encryption provided [12].

6.2.2 Authentication

As stated in section 2, authentication aims at verifying identities of communicating agents to ensure that no honest agent is impersonated by an attacker. We use correspondence events to annotate which point of the protocol has been reached. As stated earlier, a correspondence property states that if an event *e* has happened, then an event *e'* must have happened before. In this regard, to authenticate agents, we use correspondence properties to ensure that if an event *e* is "*B* accepts a run of the protocol", then an event *e'* must have happened before which is, "*A* started the run of the protocol".

This section presents the authentication of all agents in the surveillance architecture's protocol in fulfillment of requirement QR4 presented in table 4.2, which requires all agents to be authenticated before protocol messages are exchanged. We represent the agents by the following letters, let:

- A be the SO
- B be the TSA
- C be the DPA
- D be the Court
- E be the Citizen

Correspondence Events

We use the following events to authenticate the agents.
A—B (SO–TSA)

- **event** *beginAparam(pkey)*, which is used by the TSA (B) to record the belief that the initiator whose public key is supplied as parameter has commenced a run of the protocol with it.
- **event** *endAparam(pkey)*, which means that the SO (A) believes that it has successfully completed a run of the protocol with the TSA (B). This event is executed only when the SO believes it is running the protocol with the TSA i.e. when pkX = pkB where pkX is the public key A reads in to start a session with B.
- **event** *beginBparam(pkey)*, which denotes the SO's intention to initiate the protocol with an interlocutor whose public key is supplied as parameter.
- **event** *endBparam(pkey)*, which records the TSA's belief that it has successfully completed a run of the protocol with the SO. It supplies its public key pk(skB) as the parameter.

The rest of the events presented below, are similar to the ones presented above in their respective order.

A—C (SO–DPA)

- **event** *beginACparam(pkey).*
- **event** *endACparam(pkey).*
- **event** *beginCparam(pkey).*
- **event** *endCparam(pkey).*

D—A (Court–SO)

- **event** *beginDparam(pkey).*
- **event** *endDparam(pkey).*
- **event** *beginADparam(pkey).*
- **event** *endADparam(pkey).*

E—C (Citizen–DPA)

- **event** *beginEparam(pkey).*
- **event** *endEparam(pkey).*
- **event** *beginCEparam(pkey).*
- **event** *endCEparam(pkey).*

We also present a new version of the narration which includes all the agents and the messages exchanged. Let s be an observation, i be the identity of the citizen associated with observation s, and t be a timestamp.

SO	—>	TSA	: *aenc((hash(s),hash(i)),pkB)*
TSA	—>	SO	: *aenc(sign(((hash(s),hash(i)),t),sskB),pkA)*
SO	—>	DPA	: *aenc(sign(((hash(s),hash(i)),t),sskB),pkC)*
DPA	—>	SO	: *aenc(sign(sign(((hash(s),hash(i)),t),sskB),sskC),pkA)*
Court	—>	SO	: *aenc(sign(hash(i),sskD),pkA)*
SO	—>	Court	: *aenc(((s,i),sign(sign(((hash(s),hash(i)),t),sskB),sskC)),pkD)*
Citizen	—>	DPA	: *aenc(i,pkC)*
DPA	—>	Citizen	: *aenc(t,pkE)*

To summarise, we let *obs* be *(hash(s),hash(i))*. The TSA appends a timestamp t to *obs* and signs it with its key. Let the signed, timestamped *obs* be *tobs* which is *sign((obs,t),sskB)*. The SO sends *tobs* as a commitment to the DPA. The DPA then signs *tobs* and sends it back to the SO. Let the *tobs* signed by the DPA be *scom* (signed commitment) which is *sign(tobs,sskC)*. The court sends an order by sending a signed, hashed identity i for the SO to disclose records[1]. When records matching i are found, the SO sends a pair of the unhashed observation s and identity i with *scom* as *aenc(((s,i),scom),pkD)*. Since the court has the public key of both the DPA and the TSA, it would verify the unhashed observation s by computing its hash and comparing it to the hash of s in *scom*. A citizen makes a request by sending his identity i to the DPA. Since *scom* contains the timstamp t associated with this citizen, the DPA sends t to the citizen else a 'NO' answer is sent to indicate that the DPA does not have surveillance commitments related to the citizen. This leads to the following summarised version of the narration

SO	—>	TSA	: *aenc(obs,pkB)*
TSA	—>	SO	: *aenc(sign(obs,t),sskB),pkA)*
SO	—>	DPA	: *aenc(tobs,pkC)*
DPA	—>	SO	: *aenc(sign(tobs,sskC),pkA)*
Court	—>	SO	: *aenc(sign(hash(i),sskD),pkA)*
SO	—>	Court	: *aenc(((s,i),scom),pkD)*
Citizen	—>	DPA	: *aenc(i,pkC)*
DPA	—>	Citizen	: *aenc(t,pkE)*

We shall now present versions of the protocol corresponding to the incremental authentication and modeling of the agents. The first version authenticates the SO with the TSA and the last version will authenticate the Citizen with the DPA as presented in the correspondence events above. Recall that version 1.x discussed in section 6.2.1 was about secrecy, we build upon that version by adding authentication which shall be version 2.x where x shall be the iteration number. The following model versions fulfill the authentication requirement, QR4, which requires agents to be authenticated. Other specific requirements fulfilled by each version will be stated in each version.

[1]This is a simplified version as the court could issue an order based on other criteria such as a location for instance

6.2.2.1 Version 2.0: Model of the SO and the TSA

In addition to the authentication requirement (QR4), this version of the model fulfills the *integrity* requirement *QR1* which requires an observation to be timestamped.

Authentication Queries

Figure 6.9 presents correspondence events and queries between the SO and TSA for model version 2.0
The *hash* function at line 7 represents a one way hash function. Lines 26 to 30

```
7       fun hash(bitstring):bitstring.
25
26      (* Authentication queries SO–TSA*)
27      event beginBparam(pkey).
28      event endBparam(pkey).
29      event beginAparam(pkey).
30      event endAparam(pkey).
31
32      query x: pkey; inj-event(endBparam(x)) ==> inj-event
   ↪ (beginBparam(x)).
33      query x: pkey; inj-event(endAparam(x)) ==> inj-event
   ↪ (beginAparam(x)).
34
```

Figure 6.9: Model Version 2.0: Authentication Queries SO-TSA

declare the correspondence events between the SO and TSA as earlier explained. Lines 32 and 33 declare the query that will allow ProVerif to prove that the events occur in the required order. Line 32 ensures that the event *endBparam(pkey)* only takes place after *beginBparam(pkey)*, which intuitively means that the TSA ends its run of the protocol if the SO began it. Line 33 declares a similar query and ensures that the SO only ends if the TSA began the run. Injective agreement (*inj-event*) is used to ensure that for each run of the protocol on the left of ==>, there is exactly one run of the protocol on the right. This protects from replay attacks. For instance, if we left out the injective agreement and just used *event*, it would mean that the SO could obtain one timestamp for many observations hence *inj-event* ensures a one-to-one protocol run.

Secrecy Queries

Figure 6.10 presents secrecy queries

Since the standard secrecy queries of ProVerif deal with private free names, we can't directly test the secrecy of the nonces exchanged between agents, hence we declare four private free names at line 36 whose secrecy is queried at lines 38-41. The first

```
36      free secretObs, secretIdent, secretBNa, secretBNb: bitstring [private].
37
38      query attacker(secretObs);
39              attacker(secretIdent);
40              attacker(secretBNa);
41              attacker(secretBNb).
42
```

Figure 6.10: Model Version 2.0: Secrecy Queries SO-TSA

two *secretObs* and *secretIdent* represent a secret surveillance observation associated with a citizen with the secret identity *secretIdent*. For each process that acts the role of the *Initiator* of a protocol run, we test the secrecy of these two names by encrypting them with the two nonces on the initiator's side e.g. *Na* which is created by *A* and *NX* which is the nonce it receives from *B*. The last two names, *secretBNa* and *secretBNb*, are used to test the secrecy of the nonces on the *Responder's* end e.g. *B* by encrypting them with nonces *Nb*, and *NY* respectively. By having these four names, we can ascertain as to which side of the communication secrecy is not preserved [12].

SO-TSA Authentication

Figure 6.11 presents authentication steps for between the SO and the TSA.

Lines 44 to 69 declare the process macro for the SO, whose main difference from the macro in version 1.6 presented in section 6.2.1 is in lines 45 to 60 which contain the authentication code. Similarly the process macro for the TSA contains the authentication code at lines 73 to 86. The SO is the initiator for this run hence starts by reading the public key of its interlocutor at line 46. The event *beginBparam(pkX)* is registered indicating that the SO is now ready to start communication with interlocutor with the supplied key (ideally this should be the TSA but it can be any agent). The SO then creates a new nonce *Na* at line 49 and sends it to the agent with the received public key, as a challenge, together with its identity which is its public key *pk(skA)*. The TSA will receive this message at line 74. At line 75, the TSA projects on the pair after decrypting it with its secret key and saves the received nonce in variable *NY* and the public key of the initiator (which should be the SO) in the variable *pkY*. At this point the TSA can register the event that the SO began the run of the protocol with public key *pkY*, hence event *beginAparam(pkY)*. The TSA then creates its own nonce *Nb* and sends it together with the received nonce *NY* at line 78. The rest of the authentication with the SO sending back the received nonce and sending back to the TSA and both end the authentication checking that they have truly been communicating with each other i.e. TSA checks that pkY = pkA and SO checks that pkX = pkB. At lines 58, 59, 84 and 85 we test the secrecy of the nonces by using them to symmetrically encrypt the four private free names declared at line 36. This authentication scheme is used for all subsequent agents.

```
    (* SO *)
    let processA(pkB: pkey, skA: skey) =
        (*BEGIN AUTH TSA*)
        in(c, pkX: pkey);
        if pkX = pkB then
        event beginBparam(pkX);
        new Na: bitstring;
        out(c, aenc((Na, pk(skA)), pkX));
        in(c, m: bitstring);
        let (=Na, NX: bitstring,=pkX) = adec(m, skA) in
        out(c, aenc(NX, pkX));
        if pkX = pkB  then
        event endAparam(pk(skA));

        (*test secrecy of nonces*)
        out(c, senc(secretObs, Na));
        out(c, senc(secretIdent, NX));
        (*END AUTH TSA*)

    (* TSA *)
    let processB(pkA: pkey, skB: skey,sskB:sskey) =
        (*BEGIN AUTH SO*)
        in(c, m: bitstring);
        let (NY: bitstring, pkY: pkey) = adec(m, skB ) in
        event beginAparam(pkY);
        new Nb: bitstring;
        out(c, aenc((NY, Nb,pkY), pkY));
        in(c, m3: bitstring);
        if Nb = adec(m3, skB) then
        if pkY = pkA then
        event endBparam(pk(skB));

        out(c, senc(secretBNa, NY));
        out(c, senc(secretBNb, Nb));
        (*END AUTH SO*)
```

Figure 6.11: Model Version 2.0: SO-TSA Authentication

Timestamping an observation

Figure 6.12 presents the steps of each process to achieve the goal of timestamping an observation

```
61
62                 (*begin SO–TSA steps*)
63
64                 out(c, aenc(((hash(secretObs),hash(
   ↪ secretIdent)), Na,pk(skA),pkX),pkX));
65                 (*read timestamped observation*)
66                 in(c,tob:bitstring);
67                 (*tobs is signed pair of obs and ts*)
68                 let(tobs:bitstring,=pkB) = adec(tob,skA) in
69                 0.
70
87
88                 (*begin SO–TSA steps*)
89
90                 in(c,obs:bitstring);
91                 let (mb:bitstring,=NY,=pkA,=pk(skB)) = adec(
   ↪ obs,skB) in
92                 (*create timestamp*)
93                 new ts:bitstring;
94                 out(c,aenc((sign((mb,ts),sskB),pk(skB)),pkA)
   ↪ ).
95
96
97         (* Main *)
98         process
99                 new skA: skey; let pkA = pk(skA) in out(c,
   ↪ pkA);
100                new skB: skey; let pkB = pk(skB) in out(c,
   ↪ pkB);
101                new sskB:sskey; let spkB = spk(sskB) in out(
   ↪ c,spkB);
102                ( (!processA(pkB, skA)) | (!processB(pkA,
   ↪ skB,sskB)) )
103
```

Figure 6.12: Model Version 2.0: Timestamping an Observation

At line 64, the SO sends a pair of the hash of the secret observation *secretObs* and a hash of the identity of the citizen concerned which is *secretIdent*. The hash of the *secretIdent* will be used by the DPA later to identify and respond to citizen requests. The hash of *secretObs* shall be used by the Court to verify disclosed records from the SO upon a court order. Notice that in addition to the pair of the observation and identity of the citizen, the SO also sends the nonce *Na*. This is for the purpose of ensuring that each session with the TSA is *unique* to avoid replay attacks. We could have created a new nonce for the TSA to use but since the TSA already has Na, we use it instead of having to send a new one. The SO

also sends its public *pk(skA)* and the public key of its interlocutor *pkX*. The public key *pk(skA)*, will allow the TSA to check that the message it is dealing with is from the SO it previously authenticated with. *pkX* will allow the TSA to check that the message it has received is truly meant for it. The TSA will receive this request for a timestamp at line 90 and then do pattern matching at line 91. The TSA first reads the pair of *(hash(secretObs),hash(secretIdent))* in the variable *mb* and then checks that nonce in the message matches the previous one received (*=NY*), that the sender is the SO (*=pkA*) and that this message is meant for the TSA (*=pkB*)[2]. If all these patterns match, the TSA proceeds by creating a timestamp *ts* and then pairs it with the message in *mb* and then signs it with its secret signing key. The TSA then created a pair from the signed message and its public key *pk(skB)* and then encrypts this pair with the public key of the public key of the SO (*pkA*). The TSA includes its public key in this message to ensure that when the SO receives it, it can check that the message has come from the TSA, which is done by the SO at lines 66 and 68. Therefore timestamped message that the TSA finally sends to the SO is *((hash(secretObs),hash(secretIdent)),ts)*. This is the message that the SO will send to the DPA as a commitment.
Figure 6.13 shows part of ProVerif's output for version 2.0 of the model. The outputs that begin with the word RESULT show that ProVerif proves that the secrecy of all the private names is preserved and that both correspondence queries are true. This means that authentication between the SO and TSA is achieved and that the nonces they exchange are also secret.

As a summary from version 2.0:

- Agents authenticate using the Needham Schroeder protocol.
- When sending messages (not part of authentication), agents send identities to identity who it is from or who it is intended for.
- Nonces are used to ensure uniqueness of sessions to prevent reply attacks.

6.2.2.2 Version 2.1: Model of the SO and the DPA

Version 2.1 of the model fulfills the functional requirement *FR1* which requires the SO to register observations with the DPA, which together with *QR1d*, provide integrity to the observations for them to be used in court.

Authentication Queries

Figure 6.14 presents correspondence events and queries between the SO and the DPA for model version 2.1

Committing an Observation

[2]Pattern matching is syntactic sugar for assigning to a local variable and/or checking that a variable is what it should be

```
-- Query not attacker(secretObs[]); not attacker(secretIdent
   ↪ []); not attacker(secretBNa[]); not attacker(secretBNb
   ↪ [])
Completing...
Starting query not attacker(secretObs[])
RESULT not attacker(secretObs[]) is true.
Starting query not attacker(secretIdent[])
RESULT not attacker(secretIdent[]) is true.
Starting query not attacker(secretBNa[])
RESULT not attacker(secretBNa[]) is true.
Starting query not attacker(secretBNb[])
RESULT not attacker(secretBNb[]) is true.
-- Query inj-event(endAparam(x_802)) ==> inj-event(
   ↪ beginAparam(x_802))
Completing...
Starting query inj-event(endAparam(x_802)) ==> inj-event(
   ↪ beginAparam(x_802))
RESULT inj-event(endAparam(x_802)) ==> inj-event(beginAparam
   ↪ (x_802)) is true.
-- Query inj-event(endBparam(x_1607)) ==> inj-event(
   ↪ beginBparam(x_1607))
Completing...
Starting query inj-event(endBparam(x_1607)) ==> inj-event(
   ↪ beginBparam(x_1607))
RESULT inj-event(endBparam(x_1607)) ==> inj-event(
   ↪ beginBparam(x_1607)) is true.
```

Figure 6.13: ProVerif Output for Version 2.0

```
30        (* Authentication queries: SO-DPA*)
31        event beginCparam(pkey).
32        event endCparam(pkey).
33        event beginACparam(pkey).
34        event endACparam(pkey).
35
39        query x: pkey; inj-event(endCparam(x)) ==> inj-event
   ↪ (beginCparam(x)).
40        query x: pkey; inj-event(endACparam(x)) ==> inj-
   ↪ event(beginACparam(x)).
```

Figure 6.14: Model Version 2.0: Authentication Queries SO-DPA

Figure 6.14 shows the authentication events between the SO and the DPA declared at lines 30-34. Lines 39-40 declare the ProVerif queries to prove authentication. Since the authentication protocol between the SO and DPA is exactly the same as the one described for the SO and TSA—Needham Schroeder protocol, its details are left out here.

Figure 6.15 presents a model of the steps for committing an observation. Once the SO and the DPA are authenticated, the SO sends to the DPA, the timestamped observation it got from the TSA. This is done at line 93. As was the case with the TSA, the SO sends *tobs* with the nonce *Naa*, its public key *pk(skA)* and the public key of the DPA read in as *pkXA*. This serves the same purpose as mentioned before. The nonce for uniqueness of the session, the SO's public key for DPA to verify that the message is indeed from the SO and the DPA's public key for the DPA to verify that message was meant for it (line 147). Once verified, the DPA checks whether the message was timestamped and signed by the TSA (line 150) and if verified, signs and sends back the message at line 159. One aspect not included in this model is where the DPA can check how old a timestamp is, however we have provided for that in the model by ensuring that the DPA has access to the timestamp so that in an actual implementation, it would be able to compare it to the current date against some set standard of the minimum length of time for timestamps to be declared too old to be accepted. This would ensure that SO commits immediately after time stamping. In the main process, we create new secret key *skC* and secret signing key *sskC* for the DPA and then output their corresponding public keys. Below is ProVerif's output for version 2.1; we leave out ProVerif's internal representation of the processes and only show the parts related to secrecy and correspondence queries.

ProVerif output for version 2.1: SO-DPA

As may be observed from Figure 6.16 showing ProVerif's out for version 2.1 of the model, the secrecy of the private names is still preserved and all the agents are successfully authenticated.

6.2.2.3 Version 2.2: Model of the Court and the SO

This version of the model fulfills the functional requirement *FR2* which requires the the SO to disclose records to a court upon receipt of *valid* court order. The validity of the court order is ensured by letting the court sign the order. Furthermore quality requirement *QR2a* requires that the SO discloses records *only* when there's a court order. This is fulfilled by letting the SO check for the public key of the court supplied in the order (line 141 below). This ensures (confidentiality) of the records disclosed since they are disclosed only to an authorized entity. Quality requirement *QR2b* requires that the court shall verify the *integrity* of the records supplied by the court. This is done by comparing the hash of the committed observation with the hash of the disclosed observation (see lines 234-245 in Figure 6.19).
Authentication Queries

Figure 6.17 presents correspondence events and queries between the SO and the Court for version 2.2

Disclosing an Observation to the Court

Figures 6.18 (SO macro) and 6.19 (Court macro) show the model for disclosing an observation to a court upon receipt of a court order.

SO Macro

Figure 6.18 shows the process macro for the SO.
The Court as the initiator begins by sending an order at line 228. The message consists of the signed hash of the identity of the citizen being investigated, in this instance, *secretIdent*. The rest of the parts of the message are standard as has been discussed: the nonce *Nd*, the public key of court and the public key of the SO. When the SO receives this message at line 139, it reads the identity in the variable *ords* and then pattern matches to check that the message contains the nonce recently sent to the court, the public key of the court and the SO's public key. Once the patten match is successful, the SO then verifies the signature used on the order; this is to ensure integrity on the order that it is truly from the court. The *checksign* returns the actual hash of the identity which is read into variable *di*. The SO then sends a signed pair of the plain observation matching the identity, and the commitment (*reca*) received from the DPA during a commitment. This message is sent at line 149.

Court Macro

Figure 6.19 shows the process macro for the court.
The court reads it at line 231 and begins by decrypting it with its secret encryption key while pattern matching to check that the message is from the SO using *=pkA* and reading the decrypted message into variable *orec*. Next, the court checks the signature on *orec* by supplying the public signing key of the SO and reads the resulting pair of messages into variables *cobs* and *scom*; the former holding the pair of the plain observation (*secretObs*) and the hashed identity (*secretIdent*), and the latter holding the signed commitment. Next the court checks the DPA's signature on *scom* whose resulting value is read into variable *dpaCom*. Next the court splits the triple dpaCom into its actual constituents which are *hash(secretObs),hash(secretIdent),ts* which are read into variables *recObs, recIdent* and *cts* respectively (line 244). A split of *cobs* is made into the plaintext observation read as *sCobs* and the hashed identity read as *iCobs*. A hash of *sCobs* is then made and compared with the hash from the commitment which is *recObs* (line 245). If the two are matched, a check is then made to ensure that identity sent in the order matches the identity in the commitment. This ensures that records received correspond to a court order. Then court process would then save that in the database as evidence but in this model, it does nothing as indicated by the 0 process. In the main process, we add the secret encryption key (*skD*) and the secret signing key (*sskD*) of the court and then output their corresponding public keys. The court process and the SO process are instantiated appropriately with the necessary keys. ProVerif output related to version 2.2 is shown below; again we only show the query results.

ProVerif output for version 2.2: Model of Court–SO

Figure 6.20 shows the ProVerif output for version 2.2. Again, the secrecy of all private names is preserved and all agents are authenticated; we only show the results for the authentication queries.

6.2.2.4 Version 2.3: Model of the Citizen and the DPA

This version of the model fulfills the functional requirement *FR3* which requires a citizen to request the DPA if he has been under surveillance. The quality requirement *QR3* requires the DPA to access partial sets of information from the commitments, to allow it to respond to the citizen requests without compromising the secrecy of the observations. This partial set includes {hash(secretIdent),ts} which is the set of the hashed identity of the citizen and the timestamp of the observation respectively. The DPA can access this information without having knowledge of what the actual observation was since it is hashed, hence the *secrecy* property of the observation is preserved.

Authentication Queries

Figure 6.21 presents correspondence events and queries between the Citizen and the DPA for version 2.3

Requesting the DPA

Figure 6.22 shows the steps taken by the Citizen and the DPA in fulfilling the goal to satisfy a request from a citizen.
The authentication between the citizen and the DPA is not shown in Figure 6.22 but is similar to the previous pairs of agents. In this instance, the citizen is the initiator while the DPA is the responder. When the citizen and the DPA are done with the authentication, the citizen sends a request to the DPA by sending its identity at line 311 together with the nonce *Ne*, its public key *pk(skE)* and public key of the DPA *pkC*. This message is read by the DPA at line 236 where the identity received i read in variable *citIdent* and then a pattern match is made on the nonce, the public key of the citizen and that of the DPA. The DPA then checks that the hash of the identity received matches that of an existing commitment. If so, it sends the timestamp (together with the citizen's identity) of the commitment, which would inform the citizen of when he was under surveillance. This time stamp is signed by the DPA to give the citizen surety that the message is indeed from the DPA. The DPA's response is received by the citizen at line 314. The citizen decrypts the message using his secret key and pattern matches the DPA's public key to ensure that the received message is from the DPA. The citizen then checks the signature of the DPA and projects on the two components of the paired message which are the timestamp and the identity (line 316). The citizen then checks that the identity received matches his identity. In the main process we create a new secret key of the citizen *skE* at line 331 and then output its corresponding public key onto the public

channel. We also instantiate the citizen process accordingly at line 336. ProVerif's output is presented below.

ProVerif output for version 2.3: Model of Citizen–DPA

Figure 6.23 shows ProVerif output for the authentication between the Citizen and the DPA. The result is true indicating that both agents get authenticated. Again we omit output about secrecy and the previous authentication queries for the other agents. The complete output can be found in Appendix B.5.

6.2.3 Correspondence Assertions—Order of Events

This section presents version 3 of the protocol by ordering events according to the protocol narration given in section 6.2.2.

Assume we have a secret observation *secretObs* which is associated with a citizen whose identity is *secretIdent.* The following presents the list of events in the protocol.

Events

Each event takes in as parameters, the *secretIdent* associated with the *secretObs*, and the *pkey* of the agent initiating the event. Note that the *initiator* of the event may not be the same as the one *registering* the event. However when placing the events in the protocol, we place them in the process macros for the agents that register them. For instance the event *receiveCourtOrder* is initiated by the court which issues the order, but it only makes sense to place this event in the process macro for the SO who actually receives the order because it is possible that the court may issue an order which the SO never receives. Thus all the following events are deliberately prefixed with the word *receive* to show that we concentrate more on the receiving agent rather than the initiating agent. The advantage of using both the identity and the public key of the initiator is that the agents agree on both their identities and the data they are sharing, which strengthens the correspondence property as compared to when they only agree on their identities [17]

- **event** *receiveCourtOrder(bitstring,pkey)* which is registered by the SO when the Court issues an order.
- **event** *receiveOrdersRecords(bitstring,pkey)* which is registered by the Court when it receives records from the SO in response to a court order.
- **event** *receiveTSRequestFromSO(bitstring,pkey)* which is registered by the TSA when it receives a request for a timestamp from the SO.
- **event** *receiveTSFromTSA(bitstring,pkey)* which is registered by the SO when it receives a timestamped observation.
- **event** *receiveCommitment(bitstring,pkey)* which is registered by the DPA when it receives a commitment from the SO.
- **event** *receiveReceiptFromDPA(bitstring,pkey)* which is registered by the SO when it receives the signed commitment.

- **event** *receiveCitizenRequest(bitstring,pkey)* which is registered by the DPA when it receives a request from a citizen.
- **event** *receiveDPAResponse(bitstring,pkey)* which is registers by the citizen when he receives a response from the DPA.

Order of Events

We use the notation *e ==> e'* where ==> means the event on the left of ==> happens after the one on the right. Apart from the communication between the citizen and the DPA, all other events are interrelated; this is to prevent attacks such as the court sending a commitment for an observation after it has received an order for that observation. The following is the order of the events corresponding to quality requirements QR3a, QR1d, QR1a and QR2a respectively.

- receiveDPAResponse(bitstring,pkey) ==> receiveCitizenRequest(bitstring,pkey)
- receiveReceiptFromDPA(bitstring,pkey) ==> receiveCommitment(bitstring,pkey)
- receiveTSFromTSA(bitstring,pkey) ==> receiveTSRequestFromSO(bitstring,pkey)
- receiveOrdersRecords(bitstring,pkey) ==> receiveCourtOrder(bitstring,pkey)

The last three pairs of events shall be nested to enforce the order in which they should be registered. Nested events are of the form *e ==> (e' ==> e")* which means that *e* happens after *e'* which in turns happens after *e"*. Consequently it means that *e* must be last event and *e"* must be the first. The following is a nesting of the pairs of events between the SO, DPA and Court.

receiveOrdersRecords(bitstring,pkey) ==> (receiveCourtOrder(bitstring,pkey) ==> (receiveReceiptFromDPA(bitstring,pkey) ==>(receiveCommitment(bitstring,pkey) ==> (receiveTSFromTSA(bitstring,pkey) ==> receiveTSRequestFromSO(bitstring,pkey))))).

This gives the following order, beginning with the first event and ending with the last event.

1. receiveTSRequestFromSO(bitstring,pkey)
2. receiveTSFromTSA(bitstring,pkey)
3. receiveCommitment(bitstring,pkey)
4. receiveReceiptFromDPA(bitstring,pkey)
5. receiveCourtOrder(bitstring,pkey)
6. receiveOrdersRecords(bitstring,pkey)

6.2.3.1 Version 3.0 Order of Events: Citizen–DPA, SO,DPA and Court

Version 3.0 presents a model of the above events. This is in fulfillment of quality requirements QR1d which requires the DPA to sign only timestamped observations, QR2a that requires observations to be disclosed only after a court order, and QR2b that requires to check that observations were committed before being disclosed.

Correspondence Assertions

Figure 6.24 shows the correspondence events and queries that ensure that events in the protocol are excited in the desired order.

Lines 61-72 declare the correspondence events and assertions as discussed earlier.

SO Events

Figure 6.25 shows the events registered by the SO.

In the process macro for the SO, the lines of interest are 114, 143 and 175. At line 114, the SO registers the event *receiveTSFromTSA(hash(secretIdent),pkB)* to which it passes the hash of the *secretIdent* and the public key of the initiator of the event, which happens to be the TSA. This event is registered after the SO receives the timestamed observation at line 112. At line 143 it registers the event *receiveReceipt-FromDPA(hash(secretIdent),pkC)* to which passes the hash of the identity and the public key of the initiator, which is the DPA. This event is registered only after the SO receives the signed commitment at line 141. At line 175, the SO registers the event *receiveCourtOrder(di,pkD)* which is registered when the SO verifies receipt of a court order at line 174. The SO passes the received identity and the public key of the court as parameters.

TSA Events

Figure 6.26 shows the events registerred by the TSA.

The process macro for the TSA registers its event at line 206 which is *receiveTSRe-questFromSO(mbIdent,pkA)*. The TSA registers this event by passing the received hashed identity and the public ley of the SO as parameters. This event is only registered when the TSA receives a request for timestamp at line 203.

DPA Events

Figure 6.27 shows the events registered by the DPA.

The DPA registers two events, one at line 245 and the other at line 274. At line 245, it registers the event *receiveCommitment(obsIdent,pkA)* which is intiated by the SO. Recall that events are used to annotate what parts of the oriticl we have reached, therefore it is worth mentining that these events cannot be placed abitar-ily; for instance, if we placed the the event *receiveCommitment(obsIdent,pkA)* after line 249 when the DPA sends its signed commitment to the SO, the order my be distorted because the SO may receive the signed commitment and register the event *receiveReceiptFromDPA(hash(secretIdent),pkC)* before the DPA registers its event, therefore it is preferable to put the event before the sending the response to the SO. At line 274 the DPA registers the event *receiveCitizenRequest(hash(citIdent),pkE)* which is initiated by the citizen with the public supplied as parameter.

Court Events

Figure 6.28 shows the events registered by the Court.

The court registers the event *receiveOrdersRecords(recIdent,pkA)* only after it has verified the records it receives from the SO whose public key is supplied as a parameter to the event.

Citizen Events

Figure 6.29 shows the events registered by the Citizen.

Finally the citizen registers the event *receiveDPAResponse(myIdent,pkC)* when it receives a response from the the DPA with the supplied public key.

ProVerif output for version 3: Order of Events

Figure 6.30 shows ProVerif output for version 3. Only the output related to the correspondence events introduced in version 3 are included.

The results of both queries show that the order of events is preserved hence these correspondence assertions are satisfied.

In summary, this section provided a model of the architecture that fulfills requirements as expressed in Table 4.2. Table 6.1 provides a summary of which requirement was fulfilled in each version of the model. Unless otherwise explicitly stated, the quality requirement being addressed is the problem while the version implementing it provides a solution, hence some cells in the table are left blank under the problem and solution columns. QR3b which requires that the DPA shall service citizen requests without accessing secret observations was not formally verified since, in this model, the SO commits hashed observations. The full listing of the source code of the entire protocol model and ProVerif output is presented in appendix C

Table 6.1: Versioned Functional and Quality Requirements

FR	QR	Security Attribute	Increment	PROB#	SOL#
FR1	QR1a	Integrity	Version 1.0, 2.0		
	QR1b		Version 1.6	1f	1f
	QR1c		Version 1.5	1e	1e
	QR1d		Version 2.1, 3.0		
	QR1f		Version 3.0		
	QR1e	Secrecy	Version 1.1, 1.4	1a,1d	1a,1d
FR2	QR2a	Secrecy	Version 2.2		
	QR2b	Integrity	Version 2.2		
	QR2c		Version 3.0		
FR3	QR3a	Secrecy	Version 2.3		
	QR3b		(N/A)		
All	QR4	Authentication	1.2, 1.3, 2.x	1b, 1c	1b, 1c

```
    (* SO *)
    let processA(pkB: pkey,pkC:pkey, skA: skey) =
        (*begin SO–DPA steps*)
        (*tobs = (hash(obs),hash(ident),timestamp)*)
        out(c,aenc((tobs,Naa,pk(skA),pkXA),pkXA));

        (*read record from DPA*)
        in(c,rec:bitstring);
        let(reca:bitstring,=pkC) = adec(rec,skA) in
        0.
    (*DPA*)

    let processC(pkA:pkey,skC:skey,sskC:sskey,spkB:spkey
    ↪ ) =

        (*begin tasks*)
        (*wait for a commitment*)
        in(c,com:bitstring);
        let (cobs:bitstring,=NYC,=pkA,=pk(skC)) =
    ↪ adec(com,skC) in

        (*check signature of TSA to ensure cobs has
    ↪ timestamp*)
        let ckObs = checksign(cobs,spkB) in
        let (obsComit:bitstring,obsIdent:bitstring,
    ↪ obsTime:bitstring) = checksign(ckObs,spkB) in

        (*here DPA can check that timestamp is not
    ↪ too old
        e.g. by saying
        if currentDate−obsTime <= minimumLength then
    ↪  *)

        (*sign commitment and send it to SO*)
        (*ckObs is (hash(secretObs),hash(secretIdent
    ↪ ),timestamp)*)
        out(c,aenc((sign(ckObs,sskC),pk(skC)),pkA)).

    (* Main *)
    process
        new skC: skey; let pkC = pk(skC) in out(c,
    ↪ pkC);
        new sskC:sskey; let spkC = spk(sskC) in out(
    ↪ c,spkC);

        ( (!processA(pkB,pkC, skA)) | (!processB(pkA
    ↪ , skB,sskB)) |
                (!processC(pkA,skC,sskC,spkB)))
```

Figure 6.15: Model Version 2.1: Committing an Observation

```
-- Query not attacker(secretObs[]); not attacker(secretIdent
   ↪ []); not attacker(secretBNa[]); not attacker(secretBNb
   ↪ [])
Completing...
Starting query not attacker(secretObs[])
RESULT not attacker(secretObs[]) is true.
Starting query not attacker(secretIdent[])
RESULT not attacker(secretIdent[]) is true.
Starting query not attacker(secretBNa[])
RESULT not attacker(secretBNa[]) is true.
Starting query not attacker(secretBNb[])
RESULT not attacker(secretBNb[]) is true.
-- Query inj-event(endACparam(x_1346)) ==> inj-event(
   ↪ beginACparam(x_1346))
Completing...
Starting query inj-event(endACparam(x_1346)) ==> inj-event(
   ↪ beginACparam(x_1346))
RESULT inj-event(endACparam(x_1346)) ==> inj-event(
   ↪ beginACparam(x_1346)) is true.
-- Query inj-event(endCparam(x_2743)) ==> inj-event(
   ↪ beginCparam(x_2743))
Completing...
Starting query inj-event(endCparam(x_2743)) ==> inj-event(
   ↪ beginCparam(x_2743))
RESULT inj-event(endCparam(x_2743)) ==> inj-event(
   ↪ beginCparam(x_2743)) is true.
```

Figure 6.16: ProVerif Output for Version 2.1

```
        (* Authentication queries: Court(D)-SO(A) *)
        event beginDparam(pkey).
        event endDparam(pkey).
        event beginADparam(pkey).
        event endADparam(pkey).

        query x: pkey; inj-event(endDparam(x)) ==> inj-event
   ↪ (beginDparam(x)).
        query x: pkey; inj-event(endADparam(x)) ==> inj-
   ↪ event(beginADparam(x)).
```

Figure 6.17: Model Version 2.2: Authetication Queires Court-SO

```
    (* SO *)
    let processA(pkB: pkey,pkC:pkey,pkD:pkey,spkD:spkey,skA: skey,sskA:sskey) =
            ...
            (*begin steps Court-SO*)
            (*read court order*)

            in(c,ord:bitstring);
            (*ords is signed hashed secretIdent*)
            let(ords:bitstring,=Nad,=pkD,=pk(skA)) = adec(ord,skA) in

            (*check signature of court*)
            let(di:bitstring) = checksign(ords,spkD) in
            if hash(secretIdent) = di then

            (*send signed plain secretObs associated with secretIdent
            together with commitment (reca), to Court*)
            out(c,aenc((sign((secretObs,reca),sskA),pk(skA)),pkD)).

```

Figure 6.18: Model Version 2.2: SO Macro

```
    (* Court*)

    let processD(pkA:pkey,spkA:spkey,skD:skey,sskD:sskey,spkC:spkey,spkB:spkey) =
            ...
            (*begin steps Court-SO*)
            (*To ensure integrity of court order, court must sign*)
            out(c,aenc((sign(hash(secretIdent),sskD),Nd,pk(skD),pkXD),pkXD));

            (*read received records from SO*)
            in(c,crec:bitstring);
            let(orec:bitstring,=pkA) = adec(crec,skD) in

            (*check SO's signature*)
            let(cobs:bitstring,scom:bitstring) = checksign(orec,spkA) in

            (*check signature of the DPA in the SO commitment (scom)*)
            let dpaCom = checksign(scom,spkC) in

            (*check that submitted obs is what was committed
            by comparing hashes
            Recall that dpaCom is a triple of (hash(obs),hash(ident),timestamp)*)

            let(recObs:bitstring,recIdent:bitstring,cts:bitstring) = checksign(dpaCom,spkB) in
            let (sCobs:bitstring,iCobs:bitstring) = cobs in
            if hash(sCobs) = recObs then

            (*check that submitted obs is for intended Data Subject*)
            if hash(secretIdent) = recIdent then
            0.
```

Figure 6.19: Model Version 2.2: Court Macro

```
-- Query inj-event(endADparam(x_2049)) ==> inj-event(
    ↪ beginADparam(x_2049))
Completing...
Starting query inj-event(endADparam(x_2049)) ==> inj-event(
    ↪ beginADparam(x_2049))
RESULT inj-event(endADparam(x_2049)) ==> inj-event(
    ↪ beginADparam(x_2049)) is true.
-- Query inj-event(endDparam(x_4117)) ==> inj-event(
    ↪ beginDparam(x_4117))
Completing...
Starting query inj-event(endDparam(x_4117)) ==> inj-event(
    ↪ beginDparam(x_4117))
RESULT inj-event(endDparam(x_4117)) ==> inj-event(
    ↪ beginDparam(x_4117)) is true.
```

Figure 6.20: ProVerif Output for Version 2.2

```
        (* Authentication queries: Court(D)-SO(A) *)
        event beginEparam(pkey).
        event endEparam(pkey).
        event beginCEparam(pkey).
        event endCEparam(pkey).

        query x: pkey; inj-event(endEparam(x)) ==> inj-event
    ↪ (beginEparam(x)).
        query x: pkey; inj-event(endCEparam(x)) ==> inj-
    ↪ event(beginCEparam(x)).
```

Figure 6.21: Model Version 2.3: Authentication Queries Citizen-DPA

```
    (*DPA*)

    let processC(pkA:pkey,skC:skey,sskC:sskey,spkB:spkey,pkE:pkey) =
            ...
            (*begin steps with citizen*)
            in(c,request:bitstring);
            let(citIdent:bitstring,=NYE,=pkE,=pk(skC)) = adec(request,skC) in

            (*get previously saved commitment*)
            let (obsComit:bitstring,obsIdent:bitstring,obsTime:bitstring) = checksign(ckObs,spkB) in

            (*check if submitted identity matches observation*)
            if obsIdent = hash(citIdent) then
            (*send timestamp matching citizen*)
            out(c,aenc((sign((obsIdent,obsTime),sskC),pk(skC)),pkE)).

    (*Citizen*)
    let processE(pkC:pkey,skE:skey,spkC:spkey) =
            ...
            (*begin steps Citizen-DPA*)
            out(c,aenc((hash(secretIdent),Ne,pk(skE),pkXE),pkXE));

            (*read response from DPA*)
            in(c,response:bitstring);
            let(signedResponse:bitstring,=pkC) = adec(response,skE) in
            let (myTime:bitstring,myIdent:bitstring) = checksign(signedResponse,spkC) in
            if myIdent = hash(secretIdent) then
            0.
```

Figure 6.22: Model Version 2.3: Requesting the DPA

```
- Query inj-event(endCEparam(x_2742)) ==> inj-event(
   ↪ beginCEparam(x_2742))
Completing...
200 rules inserted. The rule base contains 157 rules. 16
   ↪ rules in the queue.
Starting query inj-event(endCEparam(x_2742)) ==> inj-event(
   ↪ beginCEparam(x_2742))
RESULT inj-event(endCEparam(x_2742)) ==> inj-event(
   ↪ beginCEparam(x_2742)) is true.
-- Query inj-event(endEparam(x_5524)) ==> inj-event(
   ↪ beginEparam(x_5524))
Completing...
200 rules inserted. The rule base contains 155 rules. 17
   ↪ rules in the queue.
Starting query inj-event(endEparam(x_5524)) ==> inj-event(
   ↪ beginEparam(x_5524))
RESULT inj-event(endEparam(x_5524)) ==> inj-event(
   ↪ beginEparam(x_5524)) is true.
```

Figure 6.23: ProVerif Output for Version 2.3:

```
    (*Correpondence queries to ensure correct order of
   ↪ events*)
    event receiveCourtOrder(bitstring,pkey).
    event receiveOrdersRecords(bitstring,pkey).
    event receiveTSRequestFromSO(bitstring,pkey).
    event receiveTSFromTSA(bitstring,pkey).
    event receiveCommitment(bitstring,pkey).
    event receiveReceiptFromDPA(bitstring,pkey).
    event receiveCitizenRequest(bitstring,pkey).
    event receiveDPAResponse(bitstring,pkey).

    query x: bitstring,y:pkey; inj-event(
   ↪ receiveDPAResponse(x,y)) ==> inj-event(
   ↪ receiveCitizenRequest(x,y)).
    query x: bitstring,y:pkey; inj-event(
   ↪ receiveOrdersRecords(x,y)) ==>
    (inj-event(receiveCourtOrder(x,y)) ==>
    (inj-event(receiveReceiptFromDPA(x,y)) ==>
    (inj-event(receiveCommitment(x,y)) ==>
    (inj-event(receiveTSFromTSA(x,y)) ==> inj-event(
   ↪ receiveTSRequestFromSO(x,y)))))).

```

Figure 6.24: Model Version 3.0: Correspondence Assertions (Order of Events)

```

    (* SO *)
    let processA(pkB: pkey,pkC:pkey,pkD:pkey,spkD:spkey, skA: skey,sskA:sskey) =
        (*begin SO-TSA steps*)
        (*secretObs should be hashed observation*)
        out(c, aenc(((hash(secretObs),hash(secretIdent)), Na,pk(skA),pkX),pkX));
        (*read timestamped observation*)
        in(c,tob:bitstring);
        (*tobs is signed pair of obs and ts*)
        let(tobs:bitstring,=pkB)  = adec(tob,skA) in

        event receiveTSFromTSA(hash(secretIdent),pkB);
...
        (*begin SO-DPA steps*)

        (*tobs = (hash(obs),hash(ident),timestamp)*)
        out(c,aenc((tobs,Naa,pk(skA),pkXA),pkXA));

        (*read record from DPA*)
        in(c,rec:bitstring);
        let(reca:bitstring,=pkC) = adec(rec,skA) in

        event receiveReceiptFromDPA(hash(secretIdent),pkC);

        (*end SO-DPA steps*)
        (*begin steps Court-SO*)
        (*read court order*)
        in(c,ord:bitstring);
        (*ords is signed hashed secretIdent*)
        let(ords:bitstring,=Nad,=pkD,=pk(skA)) = adec(ord,skA) in

        (*check signature of court*)
        let(di:bitstring) = checksign(ords,spkD) in

        event receiveCourtOrder(di,pkD);

        if hash(secretIdent) = di then

        (*send signed plain secretObs associated with secretIdent
        together with commitment (reca), to Court*)
        out(c,aenc((sign((secretObs,reca),sskA),pk(skA)),pkD)).
```

Figure 6.25: Model Version 3.0: SO Events

```

    (* TSA*)
    let processB(pkA: pkey, skB: skey,sskB:sskey) =
...
            (*begin tasks*)
            in(c,obs:bitstring);
            let (mb:bitstring,=NY,=pkA,=pk(skB)) = adec(obs,skB) in

            let(mbObs:bitstring,mbIdent:bitstring) = mb in

            event receiveTSRequestFromSO(mbIdent,pkA);

            (*create timestamp*)
            new ts:bitstring;
            out(c,aenc((sign((mb,ts),sskB),pk(skB)),pkA)).
```

Figure 6.26: Model Version 3.0: TSA Events)

```
    (*DPA*)

    let processC(pkA:pkey,skC:skey,sskC:sskey,spkB:spkey,pkE:pkey) =
...
          (*begin steps with SO*)
          (*wait for a commitment*)
          in(c,com:bitstring);
          let (cobs:bitstring,=NYC,=pkA,=pk(skC)) = adec(com,skC) in

          (*check signature of TSA to ensure cobs has timestamp*)
          let ckObs = checksign(cobs,spkB) in

          let (obsComit:bitstring,obsIdent:bitstring,obsTime:bitstring) = checksign(ckObs,spkB) in

          (*here DPA can check that timestamp is not too old
          e.g. by saying
          if currentDate-obsTime <= minimumLength then *)

          event receiveCommitment(obsIdent,pkA);

          (*sign commitment and send it to SO*)
          (*ckObs is (hash(secretObs),hash(secretIdent),timestamp)*)
          out(c,aenc((sign(ckObs,sskC),pk(skC)),pkA));
...
          (*begin steps with citizen*)
          in(c,request:bitstring);
          let(citIdent:bitstring,=NYE,=pkE,=pk(skC)) = adec(request,skC) in

          event receiveCitizenRequest(hash(citIdent),pkE);

          (*check if submitted identity matches observation*)
          if obsIdent = citIdent then
          (*send timestamp matching citizen*)
          out(c,aenc((sign((obsIdent,obsTime),sskC),pk(skC)),pkE)).
```

Figure 6.27: Model Version 3.0: DPA Events

```
...
    (* Court*)

    let processD(pkA:pkey,spkA:spkey,skD:skey,sskD:sskey,spkC:spkey,spkB:spkey) =
...
            (*begin steps Court-SO*)
            (*To ensure integrity of court order, court must sign*)
            out(c,aenc((sign(hash(secretIdent),sskD),Nd,pk(skD),pkXD),pkXD));

            (*read received records from SO*)
            in(c,crec:bitstring);
            let(orec:bitstring,=pkA) = adec(crec,skD) in

            (*check SO's signature*)
            let(cobs:bitstring,scom:bitstring) = checksign(orec,spkA) in

            (*check signature of the DPA in the SO commitment (scom)*)
            let dpaCom = checksign(scom,spkC) in

            (*check that submitted obs is what was committed
            by comparing hashes
            Recall that dpaCom is a triple of (hash(obs),hash(ident),timestamp)*)

            let(recObs:bitstring,recIdent:bitstring,cts:bitstring) = checksign(dpaCom,spkB) in
            if hash(cobs) = recObs then

            (*check that submitted obs is for intended Data Subject*)
            if hash(secretIdent) = recIdent then
            event receiveOrdersRecords(recIdent,pkA);
            0.
```

Figure 6.28: Model Version 3.0: Court Events

```
...

    (*Citizen*)
    let processE(pkC:pkey,skE:skey,spkC:spkey) =
...
            (*begin steps Citizen-DPA*)
            out(c,aenc((hash(secretIdent),Ne,pk(skE),pkXE),pkXE));


            (*read response from DPA*)
            in(c,response:bitstring);
            let(signedResponse:bitstring,=pkC) = adec(response,skE) in
            let (myTime:bitstring,myIdent:bitstring) = checksign(signedResponse,spkC) in
            if myIdent = hash(secretIdent) then

            event receiveDPAResponse(myIdent,pkC);
            0.
```

Figure 6.29: Model Version 3.0: Citizen Events)

```
-- Query inj-event(receiveOrdersRecords(x_2942,y_2943)) ==>
   ↪ (inj-event(receiveCourtOrder(x_2942,y_2943)) ==> (inj-
   ↪ event(receiveReceiptFromDPA(x_2942,y_2943)) ==> (inj-
   ↪ event(receiveCommitment(x_2942,y_2943)) ==> (inj-event
   ↪ (receiveTSFromTSA(x_2942,y_2943)) ==> inj-event(
   ↪ receiveTSRequestFromSO(x_2942,y_2943))))))
Completing...
200 rules inserted. The rule base contains 156 rules. 24
   ↪ rules in the queue.
Starting query inj-event(receiveOrdersRecords(x_2942,y_2943)
   ↪ ) ==> (inj-event(receiveCourtOrder(x_2942,y_2943)) ==>
   ↪  (inj-event(receiveReceiptFromDPA(x_2942,y_2943)) ==>
   ↪ (inj-event(receiveCommitment(x_2942,y_2943)) ==> (inj-
   ↪ event(receiveTSFromTSA(x_2942,y_2943)) ==> inj-event(
   ↪ receiveTSRequestFromSO(x_2942,y_2943))))))
RESULT inj-event(receiveOrdersRecords(x_2942,y_2943)) ==> (
   ↪ inj-event(receiveCourtOrder(x_2942,y_2943)) ==> (inj-
   ↪ event(receiveReceiptFromDPA(x_2942,y_2943)) ==> (inj-
   ↪ event(receiveCommitment(x_2942,y_2943)) ==> (inj-event
   ↪ (receiveTSFromTSA(x_2942,y_2943)) ==> inj-event(
   ↪ receiveTSRequestFromSO(x_2942,y_2943)))))) is true.
-- Query inj-event(receiveDPAResponse(x_6461,y_6462)) ==>
   ↪ inj-event(receiveCitizenRequest(x_6461,y_6462))
Completing...
200 rules inserted. The rule base contains 157 rules. 16
   ↪ rules in the queue.
Starting query inj-event(receiveDPAResponse(x_6461,y_6462))
   ↪ ==> inj-event(receiveCitizenRequest(x_6461,y_6462))
RESULT inj-event(receiveDPAResponse(x_6461,y_6462)) ==> inj-
   ↪ event(receiveCitizenRequest(x_6461,y_6462)) is true.
```

Figure 6.30: ProVerif Output for Version 3.0

7 Discussion

This chapter presents a discussion of the results of the evaluation of the architecture, the assumptions and limitations, feasibility of implementation, and concludes with a discussion of validity threats.

7.1 Results

The main result of this thesis is a domain model and an architecture comprising five main entities: the SO, the TSA, the DPA, the Court and the Citizen. The five were chosen in such a manner as to provide separation of concerns, however, the role of the TSA and that of the DPA could be combined i.e. the DPA could issue timestamps itself. That would reduce the entities to four. By allowing the court to perform internal verification records received from the SO, the architecture reduces the overhead associated with this cross-communication. With regard to its fulfillment of the requirements, the architecture has been evaluated by modeling and formally verifying its most critical parts in ProVerif.

The iterative approach followed in the modeling and verification process provided a step-wise verification process through which the model was built and verified incrementally. By following a process reminiscent of Test Driven Development in which developers write tests before writing code, we started out from a model where agents communicate while exchanging plaintext messages thus modeling the functional requirement and let ProVerif point out what was wrong concerning the quality requirement. This allowed us to use cryptographic primitives relevant to the problem until the security property being verified was preserved. This verification process was the most challenging part of the thesis.

This solution employs the Needham Schroeder protocol for authentication of agents. The Needham Schroeder protocol is a standard protocol for authentication and already has libraries that implement it [7], which would make it easier when developing an actual implementation of this protocol by using it as a component off the shelf, hence there was no need to use a different authentication scheme. Public key encryption is utilized (in conjunction with authentication) to provide secrecy, and digital signatures are utilized to provide integrity. Nonces are utilized to guarantee uniqueness of sessions between agents to prevent replay attacks. This is used in conjunction with injective event queries that require that for each run of the protocol by one agent, there exists exactly only one run of the protocol by its interlocutor thus ensuring authentication and integrity.

ProVerif has provided automated proofs for the preservation of the desired security properties, namely secrecy, authentication and correspondence assertions used for ordering events in the protocol.

The following requirements have been successfully modeled and verified by ProVerif:

- The SO can register surveillance observations with the DPA (FR1) and that observations can be timestamped and signed by the DPA to provide integrity (QR1a-QR1f).
- The SO can disclose records to a court upon receipt of a court order (FR2). Furthermore this only happens following a valid court order (QR2a).
- The court is able to verify records submitted to it by the SO, providing integrity for the records used as evidence in court (QR2b). The court is able to check whether a record submitted to it was signed by the DPA and timestamped by the TSA.
- A citizen is able to request the DPA for records of surveillance relating to him (FR3). The DPA is able to respond to citizen requests without compromising the secrecy of the observations from the SO (QR3a and QR3b).
- The SO is not allowed to send a commitment for an observation to the DPA after receiving a court order for that observation (QR2c).
- The model also provides for the DPA to check how old timestamps sent by the SO are by making timestamps in the commitments publicly accessible (QR1f).
- All agents authenticate each other(QR4).

Based on the foregoing, we reject H1 and H2 (see section 1.2) as this thesis has led to the design and verification of an architecture that ensures accountability. The resulting protocol of the architecture ensures that the SO has a strong incentive to commit all its observations to the DPA (because then the SO can not send an uncommitted observation to the Court without being detected) and this allows the DPA to tell the citizens whether or not they have been under surveillance. Thus the surveillance activity of the SO is controlled by accountability.

7.2 Assumptions and Limitations

This section presents the assumptions adopted during the development of the model, and the limitations of the model

7.2.1 Assumptions

This model has been built and verified based on the following assumptions (some of which maybe a repetition of what was presented in section 1.3.2.3.

- The Court verifies integrity of the records submitted by the SO based on the commitments to the DPA provided that that the original observations were not fabricated.

- Observations already exist. We do not handle the creation of observations.
- Structured meta data can be queried over observations.
- Cryptographic primitives are perfect (section 3.2.6) and can be composed (this is applied pi calculus).

7.2.2 Limitations

The following are limitations of this model.

- The model uses only one observation and one citizen; in reality the SO may send multiple records related to one or more citizens.
- The model uses the unique identity of the citizen for the court to issue an order, however other criteria could be used such as location, but are not handled by the model.
- The model does not handle internal attackers where one of the honest agents becomes compromised. Generally, compromised agents are modeled by adding an extra process called *spy* that leaks out secret keys on the public channel and then require that all other processes check that neither the initiator nor the responder is a *spy*. However, this model focuses on modeling external attackers.
- The model does not include a trusted key server which agents communicate with to access public keys. We assume there is an existing public key infrastructure managing keys.
- Timestamps are modeled as nonces since ProVerif does not support them. However, they are made public to model the fact that timestamps are not secret (This is important because you would weaken the attacker model if not done this way).
- The model does not handle strong secrecy which is modeled using observational equivalence properties.

These limitations in nowise affect the validity of the results in this thesis. See Section 7.4

7.3 Implementation Feasibility

The third hypothesis (H3) was addressed by studying the feasibility of extracting implementation code from the modeled protocol of the architecture. Two state of the art Model Driven Development (MDD) frameworks have been identified that are capable of implementing security protocols in Java; these are Spi2Java [40] and JavaSpi [6], with the latter being the latest of the two. With the use of the MDD paradigm and formal methods, these "frameworks produce security implementations with high security confidence" [43]. The frameworks allow for modeling of formal protocols based on the Dolev-Yao attacker capabilities, analysis of the resulting models to detect different kinds of logical flaws and thus provide proof for the fulfillment of the intended security properties. "Once confidence in model correctness has been reached, the models can be semi-automatically refined into Java interoperable implementations, with the guarantee that certain Dolev-Yao security properties are

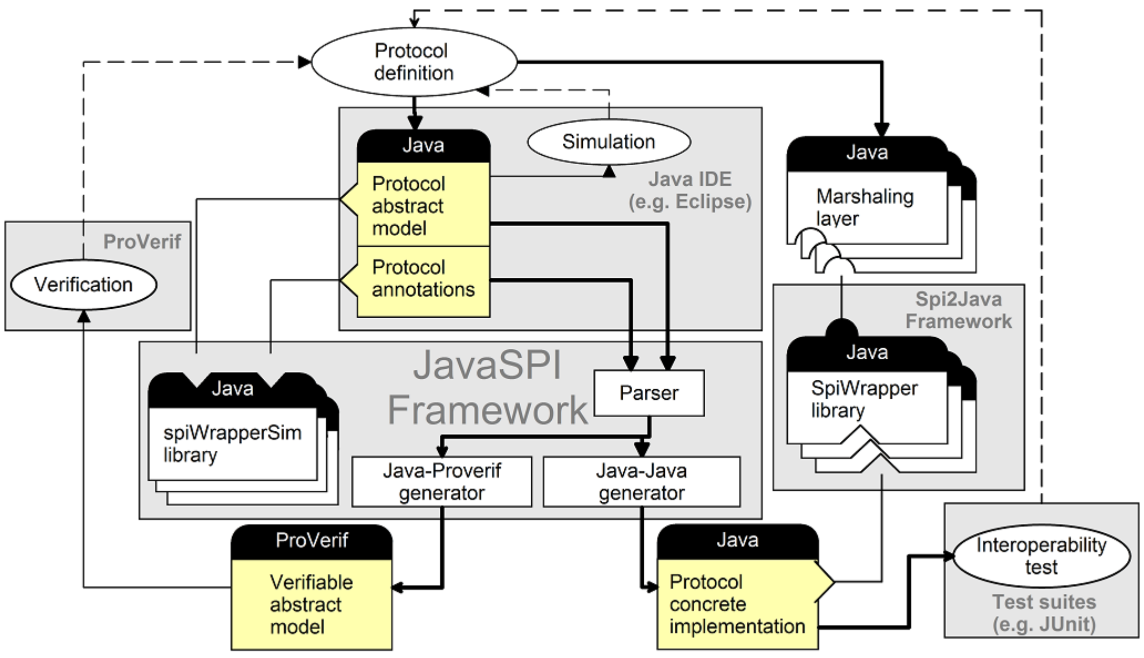


Figure 7.1: JavaSPI Framework [43]

preserved in the final implementation. This is a first step towards bridging the gap between the verified abstract formal models, and their concrete implementations.” [43]

Spi2Java was the first of the two frameworks and uses the spi-calculus for specifying protocol models. Later on, an Eclipse based graphical user interface called Spi2JavaGUI was added to Spi2Java to allow for visual modeling and graphical specification notation for models. However, in an effort to simplify the use of Spi2Java by experienced Java developers, JavaSPI was recently released. JavaSPI is a framework similar to Spi2Java and also internally uses the spi-calculus but has a different input language that is actually a subset of Java with the aid of a dedicated set of libraries. To verify models, both Spi2Java and JavaSPI internally convert to ProVerif and use it for verification. Figure 7.1 shows a worksflow diagram of JavaSPI. We here show the work flow of the JavaSPI framework since it is the more recent and better one of the two frameworks.

The key aspects to know about JavaSPI from Figure 7.1 are that JavaSPI takes in a protocol definition of a model written using a subset of the Java language. This Java based abstract model can be converted to a ProVerif verifiable abstract model which would then be verified by ProVerif, and the same Java model can also have Java code generated for its concrete implementation.

Other tools exist that convert from a concrete programing language to ProVerif; for instance FS2PV[8] is a tool that converts a protocol specification from F# to ProVerif. However such tools are not relevant to this thesis since what is required is a conversion from ProVerif’s language to a concrete implementation.

The spi-calculus[2] is an extension of the pi-calculus with predefined cryptographic primitives. The typed (applied) pi calculus which ProVerif uses to model protocols, however, allows for user-defined cryptographic primitives. Despite this limitation of the spi-calculus, it should still be possible to convert the protocol in this thesis, to the language used by Spi2Java (or JavaSPI) since all the cryptographic primitives used here do exist in the spi calculus, provided that no mistakes are introduced in this conversion step. Spi2Java cannot convert automatically from ProVerif's language to Spi2Java's language and as of this writing, no work exists in this direction; this is shown in Figure 7.2 .

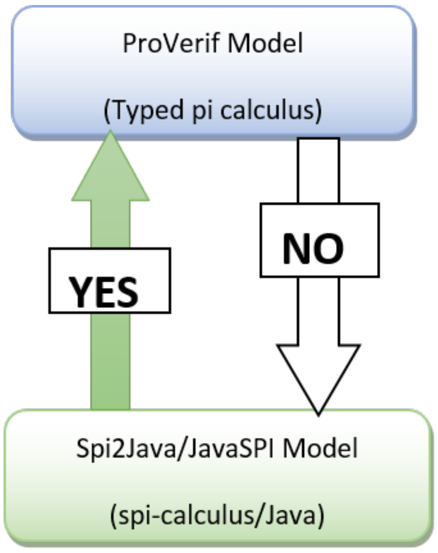


Figure 7.2: Conversion between JavaSPI and ProVerif

To produce an implementation for the protocol in this thesis using either Spi2Java or JavaSPI, a conversion of the model has to be made from ProVerif's language (typed pi calculus) to the language of the chosen target framework as shown in Figure 7.2. Automatic conversion is outside the scope of this thesis as there are currently no existing tools or techniques that do this. Therefore a manual conversion would have to be done which would involve remodeling the protocol in the target language. This might raise the question, "why not just model the protocol directly in the Spi2Java framework language?" Spi2Java, with its use of the spi-calculus is limited with regards to the definition of cryptographic primitives, hence limiting modeling capabilities, while ProVerif's language offers more flexibility. We also did not know in advance that the spi-calculus orimitives would be enough for our model. ProVerif is state of the art as most current research on security protocol verification uses it while the spi-calculus is older. The new framework–JavaSPI—was released when this thesis was already underway hence could only be considered for future work. Lastly, since this thesis focused largely on the modeling and verification of the protocol derived from the proposed architecture rather than implementation, ProVerif, with its maturity in security protocol analysis and verification, sufficed. However, as earlier stated, it is possible to convert the model of the protocol in this thesis to the language used by either Spi2Java or JavaSPI since all cryptographic primitives used in this protocol are present in the spi-calculus's set of predefined primitives, hence we reject H3 in favour of the alternative and state that it is possible to automatically extract an implementation from the modeled protocol.

7.4 Validity Threats

ProVerif uses a very powerful attacker model capturing the Dolev-Yao capabilities, which depicts an attacker who has total control of the environment and can manipulate, read and redirect messages, hence we believe the proofs offered by ProVerif do not leave out any possible attacks except those explicitly stated.The fact that in all queries performed on this model ProVerif terminated with a proof provides even more confidence since ProVerif proves over an unbounded number of sessions.

The limitation on not being able to model timestamps does not affect the validity of the proofs as timestamps basically constitute text representing time and as long as the timestamps are made public. Studies such as [4] also used an approach similar to the one used in this thesis. Neither does the lack of a key server affect the validity of the results as a key server in the model merely serves as a generalization of the model to make it more realistic. Only most critical aspects have been modeled and verified here.

This thesis focused on surveillance with regard to disclosure of records to a court. The results of this study can be generalized to situations that require similar strategies of accountability. This thesis proves that it is possible to maintain the secrecy of the SO's observations while making it accountable. The thesis does not assume the use of trusted components to ensure secrecy hence if other software engineers wanted to follow this approach to enforce accountability, they can not have worse guarantees than those provided here.

8
Conclusion

To introduce accountability in the operations of the SO, this thesis introduced a Data Protection Authority (DPA) to which the SO would be required to register its observations with a strong incentive that it could not disclose to the Court, any observation that is not register with the DPA without being detected. Hence unregistered observations have reduced utility. To ensure integrity of the observations, the SO would be required to timestamp them with a Time Stamping Authority (TSA) before sending it to the DPA. The SO would then disclose observations only upon receipt of a court order. The introduction of the DPA would allow citizens to request it if they have been under surveillance before and ultimately bring accountability to the operations of the SO.

An incremental process was followed for the modeling and verification of the protocol resulting from the proposed architecture. Starting from sequence diagrams of the architecture, protocol narrations were created which were then modeled and verified in the applied pi calculus ad typed pi calculus in ProVerif. The architecture was proven sound with regard to preserving secrecy of observations and authentication of participating agents. Furthermore, ProVerif also proved that it is not possible for the SO to cheat, for instance, by committing an observation to the DPA after receiving a court order for that observation.The SO has a strong incentive to commit all its observations to the DPA (because then the SO can not send an uncommitted observation to the Court without being detected) and this allows the DPA to tell the citizens whether or not they have been under surveillance. Thus the surveillance activity of the SO is controlled by accountability.

A better privacy guarantee is provided by ensuring that observations are disclosed to the Court only in response to a court order and also allowing citizens to check with the DPA. The secret observations are still kept secret and citizens only get to know for instance when they were under surveillance but not what was observed about them exactly. This could be extended by requiring the SO to declassify some data depending on the legal framework.

Finally this thesis provided a methodological approach that software engineers could follow to formally verify a privacy preserving architecture from Requirements, Use Cases, Domain Model, Architecture, Sequence Diagrams to the protocol Model in the applied calculus. Future work on this thesis could include modeling of internal threats as well as providing a concrete implementation of the protocol.

Bibliography

[1] Martín Abadi and Cédric Fournet. “Mobile values, new names, and secure communication”. In: *ACM SIGPLAN Notices* 36.3 (2001), pp. 104–115.

[2] Martín Abadi and Andrew D Gordon. “A calculus for cryptographic protocols: The spi calculus”. In: *Proceedings of the 4th ACM conference on Computer and communications security.* ACM. 1997, pp. 36–47.

[3] R Hevner von Alan et al. “Design science in information systems research”. In: *MIS quarterly* 28.1 (2004), pp. 75–105.

[4] Myrto Arapinis, Vincent Cheval, and Stéphanie Delaune. “Composing security protocols: from confidentiality to privacy”. In: *Principles of Security and Trust.* Springer, 2015, pp. 324–343.

[5] Alessandro Armando et al. “The AVISPA tool for the automated validation of internet security protocols and applications”. In: *Computer Aided Verification.* Springer. 2005, pp. 281–285.

[6] Matteo Avalle et al. “JavaSPI: A Framework for Security”. In: *Developing and Evaluating Security-Aware Software Systems* (2012), p. 225.

[7] Michael Backes and Birgit Pfitzmann. “A cryptographically sound security proof of the Needham-Schroeder-Lowe public-key protocol”. In: *Selected Areas in Communications, IEEE Journal on* 22.10 (2004), pp. 2075–2086.

[8] Karthikeyan Bhargavan et al. “Verified interoperable implementations of security protocols”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31.1 (2008), p. 5.

[9] Bruno Blanchet. “An efficient cryptographic protocol verifier based on Prolog rules”. In: *csfw.* IEEE. 2001, p. 82.

[10] Bruno Blanchet. “Automatic proof of strong secrecy for security protocols”. In: *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on.* IEEE. 2004, pp. 86–100.

[11] Bruno Blanchet. “Using Horn clauses for analyzing security protocols”. In: *Formal Models and Techniques for Analyzing Security Protocols* 5 (2011), pp. 86–111.

[12] Bruno Blanchet, Ben Smyth, and Vincent Cheval. *ProVerif 1.90: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial.* 2015.

[13] Huseyin Cavusoglu, Birendra Mishra, and Srinivasan Raghunathan. “The effect of internet security breach announcements on market value: Capital market reactions for breached firms and internet security developers”. In: *International Journal of Electronic Commerce* 9.1 (2004), pp. 70–104.

[14] Richard A Clarke et al. *Protecting Citizens and their Privacy.* Dec. 2013. URL: `http://www.nytimes.com/` (visited on 04/27/2016).

[15] Roger Clarke. “Internet privacy concerns confirm the case for intervention”. In: *Communications of the ACM* 42.2 (1999), pp. 60–67.

[16] CNIL. *Conclusions du contrôle des fichiers d’antécédents du ministère de l’intérieur.* June 2013. URL: `https://www.cnil.fr/sites/default/files/typo/document/Rapport_controle_des_fichiers_antecedents_judiciaires_juin_2013.pdf` (visited on 04/28/2016).

[17] Véronique Cortier and Steve Kremer. “Formal Models and Techniques for Analyzing Security Protocols: A Tutorial.” In: *Foundations and Trends in Programming Languages* 1.3 (2014), pp. 151–267.

[18] Cas JF Cremers. “The Scyther Tool: Verification, falsification, and analysis of security protocols”. In: *Computer aided verification.* Springer. 2008, pp. 414–418.

[19] Danny Dolev and Andrew C Yao. “On the security of public key protocols”. In: *Information Theory, IEEE Transactions on* 29.2 (1983), pp. 198–208.

[20] THAYER Fabrega et al. “Strand spaces: Proving security protocols correct”. In: *Journal of computer security* 7.2-3 (1999), pp. 191–230.

[21] John Faulkner. *Surveillance, Intelligence and Acountability: an Australian Story.* Oct. 2014. URL: `http://www.senatorjohnfaulkner.com.au/surveillance-intelligence-acountability-australian-story/` (visited on 04/27/2016).

[22] Oded Goldreich. *Foundations of cryptography: volume 2, basic applications.* Cambridge university press, 2009.

[23] Howarn. *The Damage of a Security Breach: Financial Institutions Face Monetary, Reputational Losses.* Apr. 2015. URL: `https://securityintelligence.com` (visited on 04/14/2016).

[24] Rivka Ladin et al. “Providing high availability using lazy replication”. In: *ACM Transactions on Computer Systems (TOCS)* 10.4 (1992), pp. 360–391.

[25] James Losey. “Surveillance of Communications: A Legitimization Crisis and the Need for Transparency”. In: *Journal of International Communication* 9 (2015), pp. 3450–3459.

[26] Gavin Lowe. “Breaking and fixing the Needham-Schroeder public-key protocol using FDR”. In: *Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 1996, pp. 147–166.

[27] David Lyon. *Surveillance studies: An overview.* Polity, 2007.

[28] Zhendong Ma et al. “Towards a Multidisciplinary Framework to Include Privacy in the Design of Video Surveillance Systems”. In: *Privacy Technologies and Policy.* Springer, 2014, pp. 101–116.

[29] Jonathan Millen and Vitaly Shmatikov. “Constraint solving for bounded-process cryptographic protocol analysis”. In: *Proceedings of the 8th ACM conference on Computer and Communications Security.* ACM. 2001, pp. 166–175.

[30] J Mitchell et al. “Undecidability of bounded security protocols”. In: *Workshop on Formal Methods and Security Protocols.* Citeseer. 1999.

[31] Moni Naor. “Bit commitment using pseudorandomness”. In: *Journal of cryptology* 4.2 (1991), pp. 151–158.

[32] T Jothi Neela and N Saravanan. “Privacy preserving approaches in cloud: a survey”. In: *Indian Journal of Science and Technology* 6.5 (2013), pp. 4531–4535.

[33] Daniel Neyland. "The Challenges of Working Out Surveillance and Accountability in Theory and Practice". In: *Managing Privacy through Accountability.* Springer, 2012, pp. 83–101.

[34] OECD. *OECD Guidelines Governing the Protection of Privacy and Trans border Flows of Personal Data.* July 2013. URL: `https://www.oecd.org/sti/ieconomy/2013-oecd-privacy-guidelines.pdf` (visited on 04/27/2016).

[35] OECD. *OECD Privacy Principles.* July 2013. URL: `http://oecdprivacy.org/` (visited on 04/27/2016).

[36] Article 29 Data Protection Working Party. *Opinion 3/2010 on the principle of accountability.* July 2010. URL: `http://ec.europa.eu/justice/policies/privacy/docs/wpdocs/2010/wp173_en.pdf` (visited on 04/27/2016).

[37] Frank A Pasquale. "Beyond Innovation and Competition: The Need for Qualified Transparency in Internet Intermediaries". In: *Available at SSRN 1686043* (2010).

[38] Siani Pearson. "Toward accountability in the cloud". In: *IEEE Internet Computing* 15.4 (2011), p. 64.

[39] Alfredo Pironti and Riccardo Sisto. *A Short Tutorial on Proverif.* Aug. 2010. URL: `http://www.cryptoforma.org.uk/old/3Sisto.pdf` (visited on 05/25/2016).

[40] Davide Pozza, Riccardo Sisto, and Luca Durante. "Spi2java: Automatic cryptographic protocol java code generation from spi calculus". In: *Advanced Information Networking and Applications, 2004. AINA 2004. 18th International Conference on.* Vol. 1. IEEE. 2004, pp. 400–405.

[41] Guttorm Sindre and Andreas L Opdahl. "Capturing security requirements through misuse cases". In: *NIK 2001, Norsk Informatikkonferanse 2001, http://www. nik. no/2001* (2001).

[42] Word Systems. *Surveillance. Types of surveillance: cameras, telephones etc.* URL: `hhttp://www.wsystems.com/news/surveillance-cameras-types.html` (visited on 05/01/2016).

[43] Politecnico di Torino. *Model-Driven Formally-Verified Implementation of Security Protocols.* 2010. URL: `http://spi2java.polito.it/` (visited on 05/01/2016).

[44] OMG UML. "2.0 specification". In: *URL http://www. omg. org/technology/-documents/formal/uml. htm* (2005).

[45] Joseph G Walls, George R Widmeyer, and Omar A El Sawy. "Building an information system design theory for vigilant EIS". In: *Information systems research* 3.1 (1992), pp. 36–59.

[46] Christoph Weidenbach. "Towards an automatic analysis of security protocols in first-order logic". In: *Automated Deduction—CADE-16.* Springer, 1999, pp. 314–328.

[47] Daniel J Weitzner et al. "Information accountability". In: *Communications of the ACM* 51.6 (2008), pp. 82–87.

[48] Stallings William. "Cryptography and network security: principles and practice". In: *Prentice-Hall, Inc* (1999), pp. 23–50.

[49] Thomas YC Woo and Simon S Lam. "Authentication for distributed systems". In: *Computer* 1 (1992), pp. 39–52.

A Appendix 1

A.1 Full Source: Version 1.0 (All Plain)

```
(* Symetric key encryption *)
type key.
fun senc(bitstring,key):bitstring.
reduc forall m:bitstring,k:key; sdec(senc(m,k),k) = m.

(* Asymetric key encryption *)
type skey.
type pkey.
fun pk(skey):pkey.
fun aenc(bitstring,pkey):bitstring.

reduc forall m:bitstring, k:skey; adec(aenc(m,pk(k)),k) = m.

(* Signing *)
type sskey.
type spkey.
fun spk(sskey):spkey.
fun sign(bitstring,sskey):bitstring.
reduc forall m: bitstring ,k:sskey; getmess(sign(m,k)) = m.

(*checksign returns m only if k matches pk(k)*)
reduc forall m:bitstring,k:sskey; checksign(sign(m,k),spk(k)) = m.

free c:channel.
(*s is an observation*)
free s:bitstring [private].

query attacker(s).

(*SO macro*)
```

```

        let clientSO() =
        out(c,s); (*send s to TSA*)
        in(c,x:bitstring); (*read timestamped s*)
        0.

        (*TSA macro*)

        let serverTSA() =
        in (c,y:bitstring);
        new t:bitstring; (*t is a time stamp which is
    ↪ modeled as a nonce*)
        out (c,(y,t)).

        (*main process*)
        process

                ((!clientSO())|(!serverTSA()))
```

B Appendix 2

B.1 Version 2.0 SO-TSA Full Source

```
free c: channel.

(* Public key encryption *)
type pkey.
type skey.

fun hash(bitstring):bitstring.

fun pk(skey): pkey.
fun aenc(bitstring, pkey): bitstring.
reduc forall x: bitstring, y: skey; adec(aenc(x, pk(
↪ y)),y) = x.

(* Signatures *)
type spkey.
type sskey.

fun spk(sskey): spkey.
fun sign(bitstring, sskey): bitstring.
reduc forall x: bitstring, y: sskey; getmess(sign(x,
↪ y)) = x.
reduc forall x: bitstring, y: sskey; checksign(sign(
↪ x,y), spk(y)) = x.

(* Shared key encryption *)
fun senc(bitstring,bitstring): bitstring.
reduc forall x: bitstring, y: bitstring; sdec(senc(x
↪ ,y),y) = x.

(* Authentication queries SO-TSA*)
event beginBparam(pkey).
event endBparam(pkey).
event beginAparam(pkey).
event endAparam(pkey).
```

```

    query x: pkey; inj-event(endBparam(x)) ==> inj-event(beginBparam(x)).
    query x: pkey; inj-event(endAparam(x)) ==> inj-event(beginAparam(x)).

    (* Secrecy queries *)
    free secretObs, secretIdent, secretBNa, secretBNb: bitstring [private].

    query attacker(secretObs);
          attacker(secretIdent);
          attacker(secretBNa);
          attacker(secretBNb).

    (* SO *)
    let processA(pkB: pkey, skA: skey) =
            (*BEGIN AUTH TSA*)
            in(c, pkX: pkey);
            if pkX = pkB then
            event beginBparam(pkX);
            new Na: bitstring;
            out(c, aenc((Na, pk(skA)), pkX));
            in(c, m: bitstring);
            let (=Na, NX: bitstring,=pkX) = adec(m, skA) in
            out(c, aenc(NX, pkX));
            if pkX = pkB  then
            event endAparam(pk(skA));

            (*test secrecy of nonces*)
            out(c, senc(secretObs, Na));
            out(c, senc(secretIdent, NX));
            (*END AUTH TSA*)

            (*begin SO-TSA steps*)

            out(c, aenc(((hash(secretObs),hash(secretIdent)), Na,pk(skA),pkX),pkX));
            (*read timestamped observation*)
            in(c,tob:bitstring);
            (*tobs is signed pair of obs and ts*)
            let(tobs:bitstring,=pkB)  = adec(tob,skA) in
            0.

    (* TSA *)
```

```
        let processB(pkA: pkey, skB: skey,sskB:sskey) =
                (*BEGIN AUTH SO*)
                in(c, m: bitstring);
                let (NY: bitstring, pkY: pkey) = adec(m, skB
    ↪ ) in
                event beginAparam(pkY);
                new Nb: bitstring;
                out(c, aenc((NY, Nb,pkY), pkY));
                in(c, m3: bitstring);
                if Nb = adec(m3, skB) then
                if pkY = pkA then
                event endBparam(pk(skB));

                out(c, senc(secretBNa, NY));
                out(c, senc(secretBNb, Nb));
                (*END AUTH SO*)

                (*begin SO-TSA steps*)

                in(c,obs:bitstring);
                let (mb:bitstring,=NY,=pkA,=pk(skB)) = adec(
    ↪ obs,skB) in
                (*create timestamp*)
                new ts:bitstring;
                out(c,aenc((sign((mb,ts),sskB),pk(skB)),pkA)
    ↪ ).


        (* Main *)
        process
                new skA: skey; let pkA = pk(skA) in out(c,
    ↪ pkA);
                new skB: skey; let pkB = pk(skB) in out(c,
    ↪ pkB);
                new sskB:sskey; let spkB = spk(sskB) in out(
    ↪ c,spkB);
                ( (!processA(pkB, skA)) | (!processB(pkA,
    ↪ skB,sskB)) )

```

B.2 Version 2.0 SO-TSA ProVerif Output

```
Process:
{1}new skA: skey;
{2}let pkA: pkey = pk(skA) in
```

```
{3}out(c, pkA);
{4}new skB: skey;
{5}let pkB: pkey = pk(skB) in
{6}out(c, pkB);
{7}new sskB: sskey;
{8}let spkB: spkey = spk(sskB) in
{9}out(c, spkB);
(
    {10}!
    {11}in(c, pkX: pkey);
    {12}if (pkX = pkB) then
    {13}event beginBparam(pkX);
    {14}new Na: bitstring;
    {15}out(c, aenc((Na,pk(skA)),pkX));
    {16}in(c, m: bitstring);
    {17}let (=Na,NX: bitstring,=pkX) = adec(m,skA) in
    {18}out(c, aenc(NX,pkX));
    {19}if (pkX = pkB) then
    {20}event endAparam(pk(skA));
    {21}out(c, senc(secretObs,Na));
    {22}out(c, senc(secretIdent,NX));
    {23}out(c, aenc(((hash(secretObs),hash(secretIdent)),Na,
       ↪ pk(skA),pkX),pkX));
    {24}in(c, tob: bitstring);
    {25}let (tobs: bitstring,=pkB) = adec(tob,skA) in
    0
) | (
    {26}!
    {27}in(c, m_67: bitstring);
    {28}let (NY: bitstring,pkY: pkey) = adec(m_67,skB) in
    {29}event beginAparam(pkY);
    {30}new Nb: bitstring;
    {31}out(c, aenc((NY,Nb,pkY),pkY));
    {32}in(c, m3: bitstring);
    {33}if (Nb = adec(m3,skB)) then
    {34}if (pkY = pkA) then
    {35}event endBparam(pk(skB));
    {36}out(c, senc(secretBNa,NY));
    {37}out(c, senc(secretBNb,Nb));
    {38}in(c, obs: bitstring);
    {39}let (mb: bitstring,=NY,=pkA,=pk(skB)) = adec(obs,skB
       ↪ ) in
    {40}new ts: bitstring;
    {41}out(c, aenc((sign((mb,ts),sskB),pk(skB)),pkA))
)
```

```
-- Query not attacker(secretObs[]); not attacker(secretIdent
   ↪ []); not attacker(secretBNa[]); not attacker(secretBNb
   ↪ [])
Completing...
Starting query not attacker(secretObs[])
RESULT not attacker(secretObs[]) is true.
Starting query not attacker(secretIdent[])
RESULT not attacker(secretIdent[]) is true.
Starting query not attacker(secretBNa[])
RESULT not attacker(secretBNa[]) is true.
Starting query not attacker(secretBNb[])
RESULT not attacker(secretBNb[]) is true.
-- Query inj-event(endAparam(x_802)) ==> inj-event(
   ↪ beginAparam(x_802))
Completing...
Starting query inj-event(endAparam(x_802)) ==> inj-event(
   ↪ beginAparam(x_802))
RESULT inj-event(endAparam(x_802)) ==> inj-event(beginAparam
   ↪ (x_802)) is true.
-- Query inj-event(endBparam(x_1607)) ==> inj-event(
   ↪ beginBparam(x_1607))
Completing...
Starting query inj-event(endBparam(x_1607)) ==> inj-event(
   ↪ beginBparam(x_1607))
RESULT inj-event(endBparam(x_1607)) ==> inj-event(
   ↪ beginBparam(x_1607)) is true.
```

B.3 Version 2.1: SO-DPA ProVerif Output

```
-- Query not attacker(secretObs[]); not attacker(secretIdent
   ↪ []); not attacker(secretBNa[]); not attacker(secretBNb
   ↪ [])
Completing...
Starting query not attacker(secretObs[])
RESULT not attacker(secretObs[]) is true.
Starting query not attacker(secretIdent[])
RESULT not attacker(secretIdent[]) is true.
Starting query not attacker(secretBNa[])
RESULT not attacker(secretBNa[]) is true.
Starting query not attacker(secretBNb[])
RESULT not attacker(secretBNb[]) is true.
-- Query inj-event(endACparam(x_1346)) ==> inj-event(
   ↪ beginACparam(x_1346))
Completing...
```

```
Starting query inj-event(endACparam(x_1346)) ==> inj-event(
    ↪ beginACparam(x_1346))
RESULT inj-event(endACparam(x_1346)) ==> inj-event(
    ↪ beginACparam(x_1346)) is true.
-- Query inj-event(endCparam(x_2743)) ==> inj-event(
    ↪ beginCparam(x_2743))
Completing...
Starting query inj-event(endCparam(x_2743)) ==> inj-event(
    ↪ beginCparam(x_2743))
RESULT inj-event(endCparam(x_2743)) ==> inj-event(
    ↪ beginCparam(x_2743)) is true.
-- Query inj-event(endAparam(x_4069)) ==> inj-event(
    ↪ beginAparam(x_4069))
Completing...
Starting query inj-event(endAparam(x_4069)) ==> inj-event(
    ↪ beginAparam(x_4069))
RESULT inj-event(endAparam(x_4069)) ==> inj-event(
    ↪ beginAparam(x_4069)) is true.
-- Query inj-event(endBparam(x_5435)) ==> inj-event(
    ↪ beginBparam(x_5435))
Completing...
Starting query inj-event(endBparam(x_5435)) ==> inj-event(
    ↪ beginBparam(x_5435))
RESULT inj-event(endBparam(x_5435)) ==> inj-event(
    ↪ beginBparam(x_5435)) is true.
```

B.4 Model Version 2.2: Court-SO ProVerif Output

```
-- Query not attacker(secretObs[]); not attacker(secretIdent
    ↪ []); not attacker(secretBNa[]); not attacker(secretBNb
    ↪ [])
Completing...
Starting query not attacker(secretObs[])
RESULT not attacker(secretObs[]) is true.
Starting query not attacker(secretIdent[])
RESULT not attacker(secretIdent[]) is true.
Starting query not attacker(secretBNa[])
RESULT not attacker(secretBNa[]) is true.
Starting query not attacker(secretBNb[])
RESULT not attacker(secretBNb[]) is true.
-- Query inj-event(endADparam(x_2049)) ==> inj-event(
    ↪ beginADparam(x_2049))
Completing...
```

```
Starting query inj-event(endADparam(x_2049)) ==> inj-event(
   ↪ beginADparam(x_2049))
RESULT inj-event(endADparam(x_2049)) ==> inj-event(
   ↪ beginADparam(x_2049)) is true.
-- Query inj-event(endDparam(x_4117)) ==> inj-event(
   ↪ beginDparam(x_4117))
Completing...
Starting query inj-event(endDparam(x_4117)) ==> inj-event(
   ↪ beginDparam(x_4117))
RESULT inj-event(endDparam(x_4117)) ==> inj-event(
   ↪ beginDparam(x_4117)) is true.
-- Query inj-event(endACparam(x_6115)) ==> inj-event(
   ↪ beginACparam(x_6115))
Completing...
Starting query inj-event(endACparam(x_6115)) ==> inj-event(
   ↪ beginACparam(x_6115))
RESULT inj-event(endACparam(x_6115)) ==> inj-event(
   ↪ beginACparam(x_6115)) is true.
-- Query inj-event(endCparam(x_8239)) ==> inj-event(
   ↪ beginCparam(x_8239))
Completing...
Starting query inj-event(endCparam(x_8239)) ==> inj-event(
   ↪ beginCparam(x_8239))
RESULT inj-event(endCparam(x_8239)) ==> inj-event(
   ↪ beginCparam(x_8239)) is true.
-- Query inj-event(endAparam(x_10317)) ==> inj-event(
   ↪ beginAparam(x_10317))
Completing...
Starting query inj-event(endAparam(x_10317)) ==> inj-event(
   ↪ beginAparam(x_10317))
RESULT inj-event(endAparam(x_10317)) ==> inj-event(
   ↪ beginAparam(x_10317)) is true.
-- Query inj-event(endBparam(x_12434)) ==> inj-event(
   ↪ beginBparam(x_12434))
Completing...
Starting query inj-event(endBparam(x_12434)) ==> inj-event(
   ↪ beginBparam(x_12434))
RESULT inj-event(endBparam(x_12434)) ==> inj-event(
   ↪ beginBparam(x_12434)) is true.
```

B.5 Model Version 2.3: Citizen-DPA ProVerif Output

```
-- Query not attacker(secretObs[]); not attacker(secretIdent
   ↪ []); not attacker(secretBNa[]); not attacker(secretBNb
   ↪ [])
Completing...
200 rules inserted. The rule base contains 157 rules. 14
   ↪ rules in the queue.
Starting query not attacker(secretObs[])
RESULT not attacker(secretObs[]) is true.
Starting query not attacker(secretIdent[])
RESULT not attacker(secretIdent[]) is true.
Starting query not attacker(secretBNa[])
RESULT not attacker(secretBNa[]) is true.
Starting query not attacker(secretBNb[])
RESULT not attacker(secretBNb[]) is true.
-- Query inj-event(endCEparam(x_2742)) ==> inj-event(
   ↪ beginCEparam(x_2742))
Completing...
200 rules inserted. The rule base contains 157 rules. 16
   ↪ rules in the queue.
Starting query inj-event(endCEparam(x_2742)) ==> inj-event(
   ↪ beginCEparam(x_2742))
RESULT inj-event(endCEparam(x_2742)) ==> inj-event(
   ↪ beginCEparam(x_2742)) is true.
-- Query inj-event(endEparam(x_5524)) ==> inj-event(
   ↪ beginEparam(x_5524))
Completing...
200 rules inserted. The rule base contains 155 rules. 17
   ↪ rules in the queue.
Starting query inj-event(endEparam(x_5524)) ==> inj-event(
   ↪ beginEparam(x_5524))
RESULT inj-event(endEparam(x_5524)) ==> inj-event(
   ↪ beginEparam(x_5524)) is true.
-- Query inj-event(endADparam(x_8319)) ==> inj-event(
   ↪ beginADparam(x_8319))
Completing...
200 rules inserted. The rule base contains 157 rules. 16
   ↪ rules in the queue.
Starting query inj-event(endADparam(x_8319)) ==> inj-event(
   ↪ beginADparam(x_8319))
RESULT inj-event(endADparam(x_8319)) ==> inj-event(
   ↪ beginADparam(x_8319)) is true.
-- Query inj-event(endDparam(x_11111)) ==> inj-event(
   ↪ beginDparam(x_11111))
Completing...
200 rules inserted. The rule base contains 155 rules. 17
   ↪ rules in the queue.
```

```
Starting query inj-event(endDparam(x_11111)) ==> inj-event(
    ↪ beginDparam(x_11111))
RESULT inj-event(endDparam(x_11111)) ==> inj-event(
    ↪ beginDparam(x_11111)) is true.
-- Query inj-event(endACparam(x_13802)) ==> inj-event(
    ↪ beginACparam(x_13802))
Completing...
200 rules inserted. The rule base contains 157 rules. 16
    ↪ rules in the queue.
Starting query inj-event(endACparam(x_13802)) ==> inj-event(
    ↪ beginACparam(x_13802))
RESULT inj-event(endACparam(x_13802)) ==> inj-event(
    ↪ beginACparam(x_13802)) is true.
-- Query inj-event(endCparam(x_16809)) ==> inj-event(
    ↪ beginCparam(x_16809))
Completing...
200 rules inserted. The rule base contains 157 rules. 19
    ↪ rules in the queue.
Starting query inj-event(endCparam(x_16809)) ==> inj-event(
    ↪ beginCparam(x_16809))
RESULT inj-event(endCparam(x_16809)) ==> inj-event(
    ↪ beginCparam(x_16809)) is true.
-- Query inj-event(endAparam(x_19580)) ==> inj-event(
    ↪ beginAparam(x_19580))
Completing...
200 rules inserted. The rule base contains 155 rules. 15
    ↪ rules in the queue.
Starting query inj-event(endAparam(x_19580)) ==> inj-event(
    ↪ beginAparam(x_19580))
RESULT inj-event(endAparam(x_19580)) ==> inj-event(
    ↪ beginAparam(x_19580)) is true.
-- Query inj-event(endBparam(x_22415)) ==> inj-event(
    ↪ beginBparam(x_22415))
Completing...
200 rules inserted. The rule base contains 157 rules. 19
    ↪ rules in the queue.
Starting query inj-event(endBparam(x_22415)) ==> inj-event(
    ↪ beginBparam(x_22415))
RESULT inj-event(endBparam(x_22415)) ==> inj-event(
    ↪ beginBparam(x_22415)) is true.
```

C
Appendix 3

C.1 Full Source code for the Protocol

```
      free c: channel.

      (* Public key encryption *)
      type pkey.
      type skey.

      fun hash(bitstring):bitstring.
      fun pk(skey): pkey.
      fun aenc(bitstring, pkey): bitstring.
      reduc forall x: bitstring, y: skey; adec(aenc(x, pk(
↪ y)),y) = x.

      (* Signatures *)
      type spkey.
      type sskey.

      fun spk(sskey): spkey.
      fun sign(bitstring, sskey): bitstring.
      reduc forall x: bitstring, y: sskey; getmess(sign(x,
↪ y)) = x.
      reduc forall x: bitstring, y: sskey; checksign(sign(
↪ x,y), spk(y)) = x.

      (* Shared key encryption *)
      fun senc(bitstring,bitstring): bitstring.
      reduc forall x: bitstring, y: bitstring; sdec(senc(x
↪ ,y),y) = x.

      (* Authentication queries: SO(A)-TSA(B) *)
      event beginBparam(pkey).
      event endBparam(pkey).
      event beginAparam(pkey).
      event endAparam(pkey).

```

```
    (* Authentication queries: SO(A)-DPA(C) *)
    event beginCparam(pkey).
    event endCparam(pkey).
    event beginACparam(pkey).
    event endACparam(pkey).

    (* Authentication queries: Court(D)-SO(A) *)
    event beginDparam(pkey).
    event endDparam(pkey).
    event beginADparam(pkey).
    event endADparam(pkey).

    (* Authentication queries: DPA(C)-Citizen(E) *)
    event beginEparam(pkey).
    event endEparam(pkey).
    event beginCEparam(pkey).
    event endCEparam(pkey).

    query x: pkey; inj-event(endBparam(x)) ==> inj-event(beginBparam(x)).
    query x: pkey; inj-event(endAparam(x)) ==> inj-event(beginAparam(x)).

    query x: pkey; inj-event(endCparam(x)) ==> inj-event(beginCparam(x)).
    query x: pkey; inj-event(endACparam(x)) ==> inj-event(beginACparam(x)).

    query x: pkey; inj-event(endDparam(x)) ==> inj-event(beginDparam(x)).
    query x: pkey; inj-event(endADparam(x)) ==> inj-event(beginADparam(x)).

    query x: pkey; inj-event(endEparam(x)) ==> inj-event(beginEparam(x)).
    query x: pkey; inj-event(endCEparam(x)) ==> inj-event(beginCEparam(x)).

    (*Correpondence queries to ensure correct order of events*)
    event receiveCourtOrder(bitstring,pkey).
    event receiveOrdersRecords(bitstring,pkey).
    event receiveTSRequestFromSO(bitstring,pkey).
    event receiveTSFromTSA(bitstring,pkey).
    event receiveCommitment(bitstring,pkey).
    event receiveReceiptFromDPA(bitstring,pkey).
    event receiveCitizenRequest(bitstring,pkey).
    event receiveDPAResponse(bitstring,pkey).

    query x: bitstring,y:pkey; inj-event(receiveDPAResponse(x,y)) ==> inj-event(receiveCitizenRequest(x,y)).
    query x: bitstring,y:pkey; inj-event(receiveOrdersRecords(x,y)) ==>
    (inj-event(receiveCourtOrder(x,y)) ==>
    (inj-event(receiveReceiptFromDPA(x,y)) ==>
    (inj-event(receiveCommitment(x,y)) ==>
    (inj-event(receiveTSFromTSA(x,y)) ==> inj-event(receiveTSRequestFromSO(x,y)))))).

    (* Secrecy queries *)
    free secretObs, secretIdent, secretBNa, secretBNb: bitstring [private].

    query attacker(secretObs);
              attacker(secretIdent);
              attacker(secretBNa);
              attacker(secretBNb).


    (* SO *)
    let processA(pkB: pkey,pkC:pkey,pkD:pkey,spkD:spkey,skA: skey,sskA:sskey) =
            (*BEGIN AUTH TSA*)
            in(c, pkX: pkey);
            event beginBparam(pkX);
            new Na: bitstring;
            out(c, aenc((Na, pk(skA)), pkX));
            in(c, m: bitstring);
            let (=Na, NX: bitstring,=pkX) = adec(m, skA) in
            out(c, aenc(NX, pkX));
            if pkX = pkB  then
            event endAparam(pk(skA));

            (*test secrecy of nonces*)
            out(c, senc(secretBNa, Na));
            out(c, senc(secretBNb, NX));
            (*END AUTH TSA*)

            (*begin SO-TSA steps*)

```

```
                (*secretObs should be hashed observation*)
                out(c, aenc(((hash(secretObs),hash(secretIdent)), Na,pk(skA),pkX),pkX));
                (*read timestamped observation*)
                in(c,tob:bitstring);
                (*tobs is signed pair of obs and ts*)
                let(tobs:bitstring,=pkB)  = adec(tob,skA) in

                event receiveTSFromTSA(hash(secretIdent),pkB);

                (*BEGIN AUTH DPA*)

                in(c, pkXA: pkey);
                event beginCparam(pkXA);
                new Naa: bitstring;
                out(c, aenc((Naa, pk(skA)), pkXA));
                in(c, ma: bitstring);
                let (=Naa, NXA: bitstring,=pkXA) = adec(ma, skA) in
                out(c, aenc(NXA, pkXA));
                if pkXA = pkC  then
                event endACparam(pk(skA));

                (*test secrecy of nonces*)
                out(c, senc(secretBNa, Naa));
                out(c, senc(secretBNb, NXA));

                (*END AUTH DPA*)

                (*begin SO-DPA steps*)

                (*tobs = (hash(obs),hash(ident),timestamp)*)
                out(c,aenc((tobs,Naa,pk(skA),pkXA),pkXA));

                (*read record from DPA*)
                in(c,rec:bitstring);
                let(reca:bitstring,=pkC) = adec(rec,skA) in

                event receiveReceiptFromDPA(hash(secretIdent),pkC);


                (*end SO-DPA steps*)

                (*BEGIN AUTH Court*)

                in(c, mad: bitstring);
                let (NYD: bitstring, pkYD: pkey) = adec(mad, skA) in
                event beginDparam(pkYD);
                new Nad: bitstring;
                out(c, aenc((NYD, Nad,pkYD), pkYD));
                in(c, md3: bitstring);
                if Nad = adec(md3, skA) then
                if pkYD = pkD then
                event endADparam(pk(skA));

                (*test secrecy of nonces*)
                out(c, senc(secretBNa, NYD));
                out(c, senc(secretBNb, Nad));

                (*END AUTH Court*)

                (*begin steps Court-SO*)
                (*read court order*)

                in(c,ord:bitstring);
                (*ords is signed hashed secretIdent*)
                let(ords:bitstring,=Nad,=pkD,=pk(skA)) = adec(ord,skA) in

                (*check signature of court*)
                let(di:bitstring) = checksign(ords,spkD) in
                event receiveCourtOrder(di,pkD);

                if hash(secretIdent) = di then

                (*send signed plain secretObs associated with secretIdent
                together with commitment (reca), to Court*)
                out(c,aenc((sign((secretObs,reca),sskA),pk(skA)),pkD)).


        (* TSA *)
        let processB(pkA: pkey, skB: skey,sskB:sskey) =
                (*BEGIN AUTH SO*)
                in(c, m: bitstring);
                let (NY: bitstring, pkY: pkey) = adec(m, skB) in
                event beginAparam(pkY);
```

```
            new Nb: bitstring;
            out(c, aenc((NY, Nb,pkY), pkY));
            in(c, m3: bitstring);
            if Nb = adec(m3, skB) then
            if pkY = pkA then
            event endBparam(pk(skB));

            out(c, senc(secretBNa, NY));
            out(c, senc(secretBNb, Nb));
            (*END AUTH SO*)

            (*begin tasks*)
            in(c,obs:bitstring);
            let (mb:bitstring,=NY,=pkA,=pk(skB)) = adec(obs,skB) in

            let(mbObs:bitstring,mbIdent:bitstring) = mb in
            event receiveTSRequestFromSO(mbIdent,pkA);

            (*create timestamp*)
            new ts:bitstring;
            out(c,aenc((sign((mb,ts),sskB),pk(skB)),pkA)).

    (*DPA*)

    let processC(pkA:pkey,skC:skey,sskC:sskey,spkB:spkey,pkE:pkey) =
            (*BEGIN AUTH SO*)
            in(c, mc: bitstring);
            let (NYC: bitstring, pkYC: pkey) = adec(mc, skC) in
            event beginACparam(pkYC);
            new Nc: bitstring;
            out(c, aenc((NYC, Nc,pkYC), pkYC));
            in(c, mc3: bitstring);
            if Nc = adec(mc3, skC) then
            if pkYC = pkA then
            event endCparam(pk(skC));

            (*test secrecy of nonces*)
            out(c, senc(secretBNa, NYC));
            out(c, senc(secretBNb, Nc));
            (*END AUTH SO*)

            (*begin tasks*)
            (*wait for a commitment*)
            in(c,com:bitstring);
            let (cobs:bitstring,=NYC,=pkA,=pk(skC)) = adec(com,skC) in

            (*check signature of TSA to ensure cobs has timestamp*)
            let ckObs = checksign(cobs,spkB) in

            let (obsComit:bitstring,obsIdent:bitstring,obsTime:bitstring) = checksign(ckObs,spkB) in

            (*here DPA can check that timestamp is not too old
            e.g. by saying
            if currentDate-obsTime <= minimumLength then *)

            event receiveCommitment(obsIdent,pkA);

            (*sign commitment and send it to SO*)
            (*ckObs is (hash(secretObs),hash(secretIdent),timestamp)*)
            out(c,aenc((sign(ckObs,sskC),pk(skC)),pkA));



            (*BEGIN AUTH Citizen*)

            in(c, ceq: bitstring);
            let (NYE: bitstring, pkYE: pkey) = adec(ceq, skC) in
            event beginEparam(pkYE);
            new Nce: bitstring;
            out(c, aenc((NYE, Nce,pkYE), pkYE));
            in(c, me3: bitstring);
            if Nce = adec(me3, skC) then
            if pkYE = pkE then
            event endCEparam(pk(skC));

            (*test secrecy of nonces*)
            out(c, senc(secretBNa, NYE));
            out(c, senc(secretBNb, Nce));

            (*END AUTH Citizen*)
```

```

                (*begin steps with citizen*)
                in(c,request:bitstring);
                let(citIdent:bitstring,=NYE,=pkE,=pk(skC)) = adec(request,skC) in
                event receiveCitizenRequest(hash(citIdent),pkE);


                (*check if submitted identity matches observation*)
                if obsIdent = citIdent then
                (*send timestamp matching citizen*)
                out(c,aenc((sign((obsIdent,obsTime),sskC),pk(skC)),pkE)).

        (* Court*)

        let processD(pkA:pkey,spkA:spkey,skD:skey,sskD:sskey,spkC:spkey,spkB:spkey) =
                (*BEGIN AUTH SO*)
                in(c, pkXD: pkey);
                event beginADparam(pkXD);
                new Nd: bitstring;
                out(c, aenc((Nd, pk(skD)), pkXD));
                in(c, md: bitstring);
                let (=Nd, NXD: bitstring,=pkXD) = adec(md, skD) in
                out(c, aenc(NXD, pkXD));
                if pkXD = pkA  then
                event endDparam(pk(skD));

                (*test secrecy of nonces*)
                out(c, senc(secretBNa, Nd));
                out(c, senc(secretBNb, NXD));
                (*END AUTH SO*)

                (*begin steps Court-SO*)
                (*To ensure integrity of court order, court must sign*)
                out(c,aenc((sign(hash(secretIdent),sskD),Nd,pk(skD),pkXD),pkXD));

                (*read received records from SO*)
                in(c,crec:bitstring);
                let(orec:bitstring,=pkA) = adec(crec,skD) in
```

```

            (*check SO's signature*)
            let(cobs:bitstring,scom:bitstring) = checksign(orec,spkA) in

            (*check signature of the DPA in the SO commitment (scom)*)
            let dpaCom = checksign(scom,spkC) in

            (*check that submitted obs is what was committed
            by comparing hashes
            Recall that dpaCom is a triple of (hash(obs),hash(ident),timestamp)*)

            let(recObs:bitstring,recIdent:bitstring,cts:bitstring) = checksign(dpaCom,spkB) in
            if hash(cobs) = recObs then

            (*check that submitted obs is for intended Data Subject*)
            if hash(secretIdent) = recIdent then
            event receiveOrdersRecords(recIdent,pkA);
            0.

    (*Citizen*)
    let processE(pkC:pkey,skE:skey,spkC:spkey) =
            (*BEGIN AUTH DPA*)
            in(c, pkXE: pkey);
            event beginCEparam(pkXE);
            new Ne: bitstring;
            out(c, aenc((Ne, pk(skE)), pkXE));
            in(c, me: bitstring);
            let (=Ne, NXE: bitstring,=pkXE) = adec(me, skE) in
            out(c, aenc(NXE, pkXE));
            if pkXE = pkC  then
            event endEparam(pk(skE));

            (*test secrecy of nonces*)
            out(c, senc(secretBNa, Ne));
            out(c, senc(secretBNb, NXE));
            (*END AUTH SO*)

            (*begin steps Citizen-DPA*)
```

```
                    out(c,aenc((hash(secretIdent),Ne,pk(skE),pkXE),pkXE));


                    (*read response from DPA*)
                    in(c,response:bitstring);
                    let(signedResponse:bitstring,=pkC) = adec(response,skE) in
                    let (myTime:bitstring,myIdent:bitstring) = checksign(signedResponse,spkC) in
                    if myIdent = hash(secretIdent) then
                    event receiveDPAResponse(myIdent,pkC);
                    0.


        (* Main *)
        process
                    new skA: skey; let pkA = pk(skA) in out(c,pkA);
                    new sskA:sskey; let spkA = spk(sskA) in out(c,spkA);
                    new skB: skey; let pkB = pk(skB) in out(c,pkB);
                    new sskB:sskey; let spkB = spk(sskB) in out(c,spkB);
                    new skC: skey; let pkC = pk(skC) in out(c,pkC);
                    new sskC:sskey; let spkC = spk(sskC) in out(c,spkC);
                    new skD: skey; let pkD = pk(skD) in out(c,pkD);
                    new sskD:sskey; let spkD = spk(sskD) in out(c,spkD);
                    new skE:skey; let pkE = pk(skE) in out(c,pkE);

                    ( (!processA(pkB,pkC,pkD,spkD,skA,sskA)) | (!processB(pkA, skB,sskB)) |
                                (!processC(pkA,skC,sskC,spkB,pkE)) |
                                (!(processD(pkA,spkA,skD,sskD,spkC,spkB))) |
                                (!(processE(pkC,skE,spkC)))
                                )

```

C.2 ProVerif Output

```
Process:
{1}new skA: skey;
{2}let pkA: pkey = pk(skA) in
{3}out(c, pkA);
{4}new sskA: sskey;
{5}let spkA: spkey = spk(sskA) in
{6}out(c, spkA);
{7}new skB: skey;
{8}let pkB: pkey = pk(skB) in
{9}out(c, pkB);
{10}new sskB: sskey;
{11}let spkB: spkey = spk(sskB) in
{12}out(c, spkB);
{13}new skC: skey;
{14}let pkC: pkey = pk(skC) in
{15}out(c, pkC);
{16}new sskC: sskey;
{17}let spkC: spkey = spk(sskC) in
{18}out(c, spkC);
{19}new skD: skey;
{20}let pkD: pkey = pk(skD) in
{21}out(c, pkD);
{22}new sskD: sskey;
{23}let spkD: spkey = spk(sskD) in
{24}out(c, spkD);
{25}new skE: skey;
{26}let pkE: pkey = pk(skE) in
{27}out(c, pkE);
(
    {28}!
    {29}in(c, pkX: pkey);
    {30}event beginBparam(pkX);
    {31}new Na: bitstring;
    {32}out(c, aenc((Na,pk(skA)),pkX));
    {33}in(c, m: bitstring);
    {34}let (=Na,NX: bitstring,=pkX) = adec(m,skA) in
    {35}out(c, aenc(NX,pkX));
    {36}if (pkX = pkB) then
    {37}event endAparam(pk(skA));
    {38}out(c, senc(secretObs,Na));
    {39}out(c, senc(secretIdent,NX));
    {40}out(c, aenc(((hash(secretObs),hash(secretIdent)),Na,
       ↪ pk(skA),pkX),pkX));
    {41}in(c, tob: bitstring);
    {42}let (tobs: bitstring,=pkB) = adec(tob,skA) in
    {43}event receiveTSFromTSA(hash(secretIdent),pkB);
    {44}in(c, pkXA: pkey);
    {45}event beginCparam(pkXA);
    {46}new Naa: bitstring;
    {47}out(c, aenc((Naa,pk(skA)),pkXA));
    {48}in(c, ma: bitstring);
    {49}let (=Naa,NXA: bitstring,=pkXA) = adec(ma,skA) in
    {50}out(c, aenc(NXA,pkXA));
    {51}if (pkXA = pkC) then
    {52}event endACparam(pk(skA));
    {53}out(c, senc(secretObs,Naa));
    {54}out(c, senc(secretIdent,NXA));
    {55}out(c, aenc((tobs,Naa,pk(skA),pkXA),pkXA));
    {56}in(c, rec: bitstring);
    {57}let (reca: bitstring,=pkC) = adec(rec,skA) in
    {58}event receiveReceiptFromDPA(hash(secretIdent),pkC);
    {59}in(c, mad: bitstring);
    {60}let (NYD: bitstring,pkYD: pkey) = adec(mad,skA) in
    {61}event beginDparam(pkYD);
    {62}new Nad: bitstring;
    {63}out(c, aenc((NYD,Nad,pkYD),pkYD));
    {64}in(c, md3: bitstring);
    {65}if (Nad = adec(md3,skA)) then
    {66}if (pkYD = pkD) then
    {67}event endADparam(pk(skA));
    {68}out(c, senc(secretBNa,NYD));
    {69}out(c, senc(secretBNb,Nad));
    {70}in(c, ord: bitstring);
    {71}let (ords: bitstring,=Nad,=pkD,=pk(skA)) = adec(ord,
      ↪ skA) in
    {72}let di: bitstring = checksign(ords,spkD) in
    {73}event receiveCourtOrder(di,pkD);
    {74}if (hash(secretIdent) = di) then
    {75}out(c, aenc((sign((secretObs,reca),sskA),pk(skA)),
      ↪ pkD))
) | (
    {76}!
    {77}in(c, m_67: bitstring);
    {78}let (NY: bitstring,pkY: pkey) = adec(m_67,skB) in
    {79}event beginAparam(pkY);
    {80}new Nb: bitstring;
    {81}out(c, aenc((NY,Nb,pkY),pkY));
    {82}in(c, m3: bitstring);
    {83}if (Nb = adec(m3,skB)) then
    {84}if (pkY = pkA) then
    {85}event endBparam(pk(skB));
    {86}out(c, senc(secretBNa,NY));
    {87}out(c, senc(secretBNb,Nb));
    {88}in(c, obs: bitstring);
    {89}let (mb: bitstring,=NY,=pkA,=pk(skB)) = adec(obs,skB
       ↪ ) in
    {90}let (mbObs: bitstring,mbIdent: bitstring) = mb in
    {91}event receiveTSRequestFromSO(mbIdent,pkA);
    {92}new ts: bitstring;
    {93}out(c, aenc((sign((mb,ts),sskB),pk(skB)),pkA))
) | (
    {94}!
    {95}in(c, mc: bitstring);
    {96}let (NYC: bitstring,pkYC: pkey) = adec(mc,skC) in
    {97}event beginACparam(pkYC);
    {98}new Nc: bitstring;
    {99}out(c, aenc((NYC,Nc,pkYC),pkYC));
    {100}in(c, mc3: bitstring);
    {101}if (Nc = adec(mc3,skC)) then
    {102}if (pkYC = pkA) then
    {103}event endCparam(pk(skC));
    {104}out(c, senc(secretBNa,NYC));
    {105}out(c, senc(secretBNb,Nc));
    {106}in(c, com: bitstring);
    {107}let (cobs: bitstring,=NYC,=pkA,=pk(skC)) = adec(com
       ↪ ,skC) in
    {108}let ckObs: bitstring = checksign(cobs,spkB) in
    {109}let (obsComit: bitstring,obsIdent: bitstring,
       ↪ obsTime: bitstring) = checksign(ckObs,spkB) in
    {110}event receiveCommitment(obsIdent,pkA);
    {111}out(c, aenc((sign(ckObs,sskC),pk(skC)),pkA));
    {112}in(c, ceq: bitstring);
    {113}let (NYE: bitstring,pkYE: pkey) = adec(ceq,skC) in
    {114}event beginEparam(pkYE);
    {115}new Nce: bitstring;
    {116}out(c, aenc((NYE,Nce,pkYE),pkYE));
    {117}in(c, me3: bitstring);
    {118}if (Nce = adec(me3,skC)) then
    {119}if (pkYE = pkE) then
    {120}event endCEparam(pk(skC));
    {121}out(c, senc(secretBNa,NYE));
    {122}out(c, senc(secretBNb,Nce));
    {123}in(c, request: bitstring);
    {124}let (citIdent: bitstring,=NYE,=pkE,=pk(skC)) = adec
       ↪ (request,skC) in
    {125}event receiveCitizenRequest(hash(citIdent),pkE);
    {126}if (obsIdent = citIdent) then
    {127}out(c, aenc((sign((obsIdent,obsTime),sskC),pk(skC))
      ↪ ,pkE))
) | (
    {128}!
    {129}in(c, pkXD: pkey);
    {130}event beginADparam(pkXD);
    {131}new Nd: bitstring;
    {132}out(c, aenc((Nd,pk(skD)),pkXD));
    {133}in(c, md: bitstring);
    {134}let (=Nd,NXD: bitstring,=pkXD) = adec(md,skD) in
    {135}out(c, aenc(NXD,pkXD));
    {136}if (pkXD = pkA) then
    {137}event endDparam(pk(skD));
    {138}out(c, senc(secretObs,Nd));
    {139}out(c, senc(secretIdent,NXD));
    {140}out(c, aenc((sign(hash(secretIdent),sskD),Nd,pk(skD
      ↪ ),pkXD),pkXD));
    {141}in(c, crec: bitstring);
    {142}let (orec: bitstring,=pkA) = adec(crec,skD) in
    {143}let (cobs_68: bitstring,scom: bitstring) =
      ↪ checksign(orec,spkA) in
    {144}let dpaCom: bitstring = checksign(scom,spkC) in
    {145}let (recObs: bitstring,recIdent: bitstring,cts:
      ↪ bitstring) = checksign(dpaCom,spkB) in
    {146}if (hash(cobs_68) = recObs) then
    {147}if (hash(secretIdent) = recIdent) then
    {148}event receiveOrdersRecords(recIdent,pkA)
) | (
    {149}!
    {150}in(c, pkXE: pkey);
    {151}event beginCEparam(pkXE);
    {152}new Ne: bitstring;
    {153}out(c, aenc((Ne,pk(skE)),pkXE));
    {154}in(c, me: bitstring);
    {155}let (=Ne,NXE: bitstring,=pkXE) = adec(me,skE) in
    {156}out(c, aenc(NXE,pkXE));
    {157}if (pkXE = pkC) then
    {158}event endEparam(pk(skE));
    {159}out(c, senc(secretObs,Ne));
    {160}out(c, senc(secretIdent,NXE));
    {161}out(c, aenc((hash(secretIdent),Ne,pk(skE),pkXE),
      ↪ pkXE));
    {162}in(c, response: bitstring);
    {163}let (signedResponse: bitstring,=pkC) = adec(
      ↪ response,skE) in
    {164}let (myTime: bitstring,myIdent: bitstring) =
       ↪ checksign(signedResponse,spkC) in
    {165}if (myIdent = hash(secretIdent)) then
    {166}event receiveDPAResponse(myIdent,pkC)
)

-- Query not attacker(secretObs[]); not attacker(secretIdent
   ↪ []); not attacker(secretBNa[]); not attacker(secretBNb
   ↪ [])
Completing...
200 rules inserted. The rule base contains 157 rules. 14
   ↪ rules in the queue.
Starting query not attacker(secretObs[])
RESULT not attacker(secretObs[]) is true.
Starting query not attacker(secretIdent[])
RESULT not attacker(secretIdent[]) is true.
Starting query not attacker(secretBNa[])
RESULT not attacker(secretBNa[]) is true.
Starting query not attacker(secretBNb[])
RESULT not attacker(secretBNb[]) is true.
-- Query inj-event(receiveOrdersRecords(x_2942,y_2943)) ==>
   ↪ (inj-event(receiveCourtOrder(x_2942,y_2943)) ==> (inj-
   ↪ event(receiveReceiptFromDPA(x_2942,y_2943)) ==> (inj-
   ↪ event(receiveCommitment(x_2942,y_2943)) ==> (inj-event
   ↪ (receiveTSFromTSA(x_2942,y_2943)) ==> inj-event(
   ↪ receiveTSRequestFromSO(x_2942,y_2943))))))
Completing...
200 rules inserted. The rule base contains 156 rules. 24
   ↪ rules in the queue.
Starting query inj-event(receiveOrdersRecords(x_2942,y_2943)
   ↪ ) ==> (inj-event(receiveCourtOrder(x_2942,y_2943)) ==>
   ↪  (inj-event(receiveReceiptFromDPA(x_2942,y_2943)) ==>
   ↪ (inj-event(receiveCommitment(x_2942,y_2943)) ==> (inj-
   ↪ event(receiveTSFromTSA(x_2942,y_2943)) ==> inj-event(
   ↪ receiveTSRequestFromSO(x_2942,y_2943))))))
RESULT inj-event(receiveOrdersRecords(x_2942,y_2943)) ==> (
   ↪ inj-event(receiveCourtOrder(x_2942,y_2943)) ==> (inj-
   ↪ event(receiveReceiptFromDPA(x_2942,y_2943)) ==> (inj-
   ↪ event(receiveCommitment(x_2942,y_2943)) ==> (inj-event
   ↪ (receiveTSFromTSA(x_2942,y_2943)) ==> inj-event(
   ↪ receiveTSRequestFromSO(x_2942,y_2943)))))) is true.
-- Query inj-event(receiveDPAResponse(x_6461,y_6462)) ==>
   ↪ inj-event(receiveCitizenRequest(x_6461,y_6462))
Completing...
200 rules inserted. The rule base contains 157 rules. 16
   ↪ rules in the queue.
```

```
Starting query inj-event(receiveDPAResponse(x_6461,y_6462))
   ↪ ==> inj-event(receiveCitizenRequest(x_6461,y_6462))
RESULT inj-event(receiveDPAResponse(x_6461,y_6462)) ==> inj-
   ↪ event(receiveCitizenRequest(x_6461,y_6462)) is true.
-- Query inj-event(endCEparam(x_9321)) ==> inj-event(
   ↪ beginCEparam(x_9321))
Completing...
200 rules inserted. The rule base contains 157 rules. 16
   ↪ rules in the queue.
Starting query inj-event(endCEparam(x_9321)) ==> inj-event(
   ↪ beginCEparam(x_9321))
RESULT inj-event(endCEparam(x_9321)) ==> inj-event(
   ↪ beginCEparam(x_9321)) is true.
-- Query inj-event(endEparam(x_12320)) ==> inj-event(
   ↪ beginEparam(x_12320))
Completing...
200 rules inserted. The rule base contains 155 rules. 17
   ↪ rules in the queue.
Starting query inj-event(endEparam(x_12320)) ==> inj-event(
   ↪ beginEparam(x_12320))
RESULT inj-event(endEparam(x_12320)) ==> inj-event(
   ↪ beginEparam(x_12320)) is true.
-- Query inj-event(endADparam(x_15390)) ==> inj-event(
   ↪ beginADparam(x_15390))
Completing...
200 rules inserted. The rule base contains 157 rules. 16
   ↪ rules in the queue.
Starting query inj-event(endADparam(x_15390)) ==> inj-event(
   ↪ beginADparam(x_15390))
RESULT inj-event(endADparam(x_15390)) ==> inj-event(
   ↪ beginADparam(x_15390)) is true.
-- Query inj-event(endDparam(x_18389)) ==> inj-event(
   ↪ beginDparam(x_18389))
Completing...
200 rules inserted. The rule base contains 155 rules. 17
   ↪ rules in the queue.
Starting query inj-event(endDparam(x_18389)) ==> inj-event(
   ↪ beginDparam(x_18389))
RESULT inj-event(endDparam(x_18389)) ==> inj-event(
   ↪ beginDparam(x_18389)) is true.
-- Query inj-event(endACparam(x_21280)) ==> inj-event(
   ↪ beginACparam(x_21280))
Completing...
200 rules inserted. The rule base contains 157 rules. 16
   ↪ rules in the queue.
```

```
Starting query inj-event(endACparam(x_21280)) ==> inj-event(
    ↪ beginACparam(x_21280))
RESULT inj-event(endACparam(x_21280)) ==> inj-event(
    ↪ beginACparam(x_21280)) is true.
-- Query inj-event(endCparam(x_24595)) ==> inj-event(
    ↪ beginCparam(x_24595))
Completing...
200 rules inserted. The rule base contains 157 rules. 19
    ↪ rules in the queue.
Starting query inj-event(endCparam(x_24595)) ==> inj-event(
    ↪ beginCparam(x_24595))
RESULT inj-event(endCparam(x_24595)) ==> inj-event(
    ↪ beginCparam(x_24595)) is true.
-- Query inj-event(endAparam(x_27566)) ==> inj-event(
    ↪ beginAparam(x_27566))
Completing...
200 rules inserted. The rule base contains 155 rules. 15
    ↪ rules in the queue.
Starting query inj-event(endAparam(x_27566)) ==> inj-event(
    ↪ beginAparam(x_27566))
RESULT inj-event(endAparam(x_27566)) ==> inj-event(
    ↪ beginAparam(x_27566)) is true.
-- Query inj-event(endBparam(x_30612)) ==> inj-event(
    ↪ beginBparam(x_30612))
Completing...
200 rules inserted. The rule base contains 157 rules. 19
    ↪ rules in the queue.
Starting query inj-event(endBparam(x_30612)) ==> inj-event(
    ↪ beginBparam(x_30612))
RESULT inj-event(endBparam(x_30612)) ==> inj-event(
    ↪ beginBparam(x_30612)) is true.
```