An analysis of security protocols for lightweight systems

by

Martha Ndeyapeuomagano Kamkuemah

Dissertation presented for the degree of Doctor of Philosophy in Mathematics in the Faculty of Science at Stellenbosch University

Supervisor: Prof. J.W. Sanders

April 2022

Declaration

By submitting this dissertation electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: April 2022 .....

Abstract

An analysis of security protocols for lightweight systems

M.N. Kamkuemah

Department of Mathematical Sciences
University of Stellenbosch
Private Bag X1, Matieland 7602 , South Africa.

Dissertation: PhD

April 2022

Security is hard to maintain in distributed systems especially for communicating agents restricted to lightweight computations, as in the Internet of Things, which struggle to implement strong cryptographic security. A methodology is developed for specifying and reasoning algebraically about security in such systems which combines epistemic logic and a state-based formalism. The knowledge modality K is used to define authentication and secrecy in terms of what each agent knows. Operations are defined as state transitions. Having gained confidence in our methodology by applying it to the benchmark case studies Needham-Schroeder and Diffie-Hellman protocols, we then apply it to the contemporary examples Signal and Long-Range Wide-Area Network protocols. A mitigation is proposed and verified for a Long-Range Wide-Area Network.

Uittreksel

An analysis of security protocols for lightweight systems

M.N. Kamkuemah

Department of Mathematical Sciences
University of Stellenbosch
Private Bag X1, Matieland 7602 , South Africa.

Proefskrif: PhD

April 2022

Sekuriteit is moeilik om te handhaaf in verspreide stelsels, veral vir kommunikasie-agente met beperkte berekenings vermoë, soos Internet van Dinge, wat sukkel om sterk kriptografiese sekuriteit te implimenteer. 'n Metodologie word ontwikkel vir die spesifikasie en algebraïes redenering angaande sekuriteit vir sulke sisteme. Hierdie metodologie maak van epistemiese logika en 'n staat gebaseerde formalisme gebruik. Die kennismodaliteit K word gebruik om verifikasie en geheimhouding te definieer in terme van wat elke agent weet. Operasies word as staatsoorgange gedefinieer. Nadat vertroue in die metodologie verkry word deur dit op die maatstaf gevallestudies van die Needham-Schroeder- en Diffie-Hellman protokolle toe te pas, word dit vervolgens op die hedendaagse voorbeelde van Sein en Langafstand Wye-

area netwerk protokolle toegepas. ‘n Versagting word vir ‘n Langafstand Wye-area netwerk voorgestel en geverifieer.

Acknowledgement

I want to thank my family for their unconditional support.

A special thank you to Brink van der Merwe from Computer Science, Stellenbosch University for translating the thesis abstract into Afrikaans.

Finally I want to thank my supervisor for his guidance while working on this thesis.

Dedication

To my mother, whose courage and strength have made me the person I am: "The world asks of us only the strength we have and we give it. Then it asks more, and we give it." – The Weighing by Jane Hirshfield.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides the context of this thesis, our methodology, questions tackled and summarises related work.

1.1 Historical context

When programming a single computer dominated computation, the major difficulty of correctness involved nontermination. Of course computing the right output was essential, but with nontermination the programmer was unable to distinguish, by program execution, delayed output from none at all, which has its theoretical foundation in the Halting Problem: in general there is no algorithm for detecting nontermination. Distributed systems now provide mainstream computation, from multiprogramming (Eindhoven 1960's), Aloha-net (Hawaii 1971), Arpanet (late 1970's), the Internet, the Web (1991), to the Internet of Things (since the late 90's). Reflecting the distinction between uniprocessor and distributed systems the word "algorithm" is used for the former and "protocol" for the latter.

With protocols in distributed systems come new difficulties: livelock, privacy, security, channel corruption, fault-tolerance, self-stabilisation, consistency, etc. Even input-output correctness, called functional correctness, is more subtle because of the coordination used to attain the result.

From the 1980's security has been seen as a nonfunctional property, in which it resembles complexity. A functional specification describes the desired input-output behaviour without regard for the efficiency, or computational complexity, with which it is to be reached in the implementation. This thesis explores the extent to which privacy and security *can* be specified and reasoned about as functional properties.

1.2 Domain information

The development of distributed systems was captured in the mid 1980's by the OSI-ISO 7-layer model [85]. At that time the internet used the TCP/IP protocol [80, 70] for communication at the transport layer (layer 4, where layer 1 is the physical layer, layer 2 is the data link layer, layer 3 is the network layer, layer 5 is the session layer, layer 6 is the presentation layer, and layer 7 is the application layer). So we begin by addressing privacy and security of TCP, which is important because it is used by higher layers.

With the advent of smart devices, the ISO model has needed revision [85]. Smart devices like sensors and actuators designed for specific tasks do not operate at the layers of the ISO model due to limitations in their processing power, memory, bandwidth, and storage capacity. Contemporary networks involving the Internet of Things (IoT) use either a 3-layer or 5-layer architecture [78]. For instance, a device using the Long Range Wide Area Network (LoRaWAN) [62] communication protocol operates at 3 layers of the ISO model, including the physical layer for transmitting messages over physical media; the network layer for routing messages; and finally the application layer for message formatting and human interaction. The 3-layer architecture has advantages and disadvantages. It is cost-effective for manufacturers to build smart devices that use fewer layers but these designs have security issues. Devices are often produced with basic-level security like default passwords. Furthermore, according to Suo *et al.* [83] some communication protocols for these networks lack security by design.

1.3 Methodology

Since the 1980's Formal Methods have been used in the hope of taming the complexities of distributed systems. Functional properties have been expressed to specify and describe system designs, and a concept of refinement was developed to connect the two. For interactive systems safety and liveness were used. But it was not seen how to capture privacy and security functionally, so those remained nonfunctional.

In this thesis a distributed system is seen as a collection of agents with disjoint state spaces, communicating by message passing. This differs from an alternative model which uses shared variables. It makes it easier for us to express and reason about which agent can know which property. Privacy and security properties have a natural knowledge interpretation, for example, what legitimate agents know about a property. For example, mutual authentication can be expressed as *both agents know they both know a certain value* while secrecy can be expressed as *an intruder does not know certain information* [20]. Knowledge is represented with the modal operator K_i, where the subscript indicates the agent. Formula $K_i\,\phi$ reads "agent i knows fact ϕ". These knowledge statements form part of epistemic logic which has sound and complete semantics defined in general by Kripke [53] and specifically by Fagin *et al.* [27, 26, 28], and Chandy and Misra [13]. Nested knowledge operators are a recurring theme in this thesis necessary for expressing knowledge gained as agents communicate. The above interpretation assumes the agents possess the knowledge. In some instances we are interested in reasoning about knowledge that an agent does not possess but knows another agent possesses. This is the concept of zero-knowledge.

Finally, a state-based notation (in particular Z) [82] is used to specify security protocols as discrete dynamical systems [45] with state and operations on state. We combine the Z specifications with epistemic statements to reason about correctness and show protocols are secure. Sometimes the cryptographic notation of a protocol is also used to reason about security.

The nonfunctional property of efficiency has evolved with the development of distributed systems. For uniprocessors it involved only space-time complexity, but now it involves much more: also number of messages, firewall activity, etc. It will be important when we consider designs for IoT that they be computationally realistic, since those devices have limited computational capability. An example is a children's toy that is capable of taking video and connecting to the internet (see for example "The Cayla doll" [65]). Privacy mitigations for flaws in TCP may be far from appropriate for LoRaWAN.

Privacy and security have been found difficult to maintain in distributed systems particularly because new attack opportunities are regularly discovered (see [32, 65, 40, 44]), in particular – as we have seen – IoT presents new opportunities for attack. In view of the activity on the internet it is not easy to identify attacks not already covered there, so we shall be satisfied with describing existing attacks.

1.4 Research aims

In order to provide efficient and trusted services for devices such as sensors in the IoT, there is a need to preserve the privacy and security of agents whose computational resources are limited and whose privacy is vulnerable to attack. This thesis explores the extent to which privacy and security goals of IoT protocols can be expressed and reasoned about algebraically at the design level, without having to resort to reasoning at the level of code. The result is a calculus that can be harnessed during design and not as an afterthought. To meet this goal, the thesis uses epistemic logic to specify security properties and to analyse protocol behaviour. The work specifies security protocols using a state-based approach and analyses their behaviour using epistemic laws. This approach is applied to the benchmarks: the Needham-Schroeder Protocol and the Diffie-Hellman Key-Exchange Protocol. It is then applied to Lamport's One-Time Password-Authentication Scheme. Satisfied with its applicability we then apply the approach to protocols designed for lightweight distributed systems. These include the instant messaging protocol Sig-

nal, and an IoT protocol called LoRaWAN.

In summary, this work addresses the following research question:

> *To what extent can privacy and security of protocols designed for lightweight distributed systems be specified and reasoned about algebraically using epistemic logic and a state-based approach?*

To answer this question, this thesis makes the following contributions. It provides novel and precise epistemic definitions of privacy, mutual authentication and secrecy. It then specifies the behaviour of various protocols using the Z language while ensuring that these specifications are correct and consistent. This approach is applied to various lightweight security protocols and presents results in the form of proofs about security, based on certain assumptions about their cryptographic primitives. We also provide theoretical evaluations on the efficiency of the solutions suggested to mitigate weaknesses in one of the lightweight distributed protocols and a new proposal is similarly treated. This approach is novel in the context of IoT.

1.5 Related work

Security comprises many properties; here we consider authentication and secrecy and ways to give precise definitions for them. Techniques include process algebra [42], model-checking methods [72], verification tools [19], and logics [2, 12, 84].

Glasgow *et al.* [33] use modal logic to specify secrecy. Their work is related because it also uses the knowledge modality to specify secrecy in terms of what each agent knows. Our work extends the definition of secrecy and includes a definition for authentication.

Lowe [57] uses process algebra to specify various forms of authentication. We use Lowe's agreement definition which states there is a one-one relationship between

values shared by agents after completing a protocol run. We express this form of authentication in terms of who knows what.

In the case of authentication, Needham and Schroeder [67] use BAN logic [12] to show that the Needham-Schroeder authentication protocol does indeed satisfy authentication. Their protocol design was later shown by Lowe [56] to be susceptible to a man-in-the-middle attack and therefore did not satisfy authentication. Lowe patched the protocol, now called Needham-Schroeder-Lowe [56], and justified his patch by modelling an intruder in a process algebra called Communicating Sequential Processes (CSP). We analyse the man-in-the-middle attack of Lowe in epistemic logic.

Without considering security or privacy, Smith [80] used IO automata to prove that TCP implemented its functional properties. His method used invariant assertions and refinement. By specifying TCP in terms of bounded and unbounded sequence number generation, he showed correctness of an experimental implementation of TCP called T/TCP. We also use a state-based formalism, the language Z, whose schema bodies act as invariants. Our TCP is an abstraction of that considered by Smith and concentrates on security properties.

Lamport [54] used a one-way function mapping to achieve authentication. A one-way function maps some set of bitstring into itself such that given x, $f(x)$ is easy to computing but given y, if is infeasible to compute x such that $y = f(x)$. Lamport showed correctness of this protocol by implementing a one-time password authentication protocol. Our work studies the protocol using our approach as a zero-knowledge protocol.

More recent work that gives definitions of authentication and secrecy and applies them to new protocols is that of Cohn-Gordon *et al.* [17] and Frosch *et al.* [31]. Cohn-Gordon *et al.* [17] defined predicates for authentication and secrecy and applied them to the Signal protocol [62] designed for private communications between users. They showed the protocol is secure under certain cryptographic assumptions.

Similarly, Frosch *et al.* [31], showed that TextSecure, the predecessor of Signal, is secure. Our approach is similar to that of Cohn-Gordon. Both works create an abstract model of the protocol in question. However, our proofs use epistemic predicates and require explicit reasoning about an intruder while proofs by Cohn-Gordon *et al.* [17] are in random oracle setting which reasons about security based on hardness assumptions about the cryptographic algorithms used by the protocol.

Other work that is relevant to this research but which used model-checking to give precise definitions of authentication is that of Eldefrawy *et al.* [24]. Among the security properties the authors defined a form of authentication called non-injective agreement. Lowe [57] describes non-injective agreement as a form of authentication where two parties agree on a set of data variables whenever a party acting as initiator completes a run of the protocol, apparently with a responder, and vice versa. They build an abstract model of LoRaWAN using the verification tool Scyther consisting of these security claims. Again, our work uses a state-based approach to build an abstraction of LoRaWAN and reason about its security. In both cases our more abstract approach applies at the design stage, and confers the ability to reason about a range of implementations at once.

Stronger models of privacy which we have not needed to resort to, include the Shadow semantics (work of Morgan *et al.* [64, 63]) and other quantitative measures of information flow (work of Geoff Smith *et al.* [79]).

For a survey of further communications protocols to which the techniques here can be applied see Clark and Jacob [16].

Publications arising from this thesis are:

[47] A conference paper (to appear in proceedings) summarising Chapter 6, Analysis of Signal. However no state transitions are included (to keep the paper Z free) and so the formalisation of Signal there is treated with a light touch. In the thesis numerical experiments are included.

[46] A conference paper (to appear in proceedings) summarising Chapter 7, Vulnerability in LoRaWAN. Again state transitions are not formalised there, for the same reason.

[48] A conference paper (submitted) summarising Chapter 4, Zero Knowledge, both in general and applied authentication. Lamport's One-Time Authentication Scheme is treated as a stand alone study.

[45] A workshop talk explaining how discrete transition systems of the type encountered in this thesis are specified and implemented as abstract data types, and how this is analogous to a differential equation specifying a solution for a smooth, rather than discrete, system.

1.6 Organisation

The rest of the thesis is organised as follows. Notation for a state-based formalism and epistemic laws are given in Chapter 2. Epistemic logic is used in Chapter 3 to produce definitions for privacy, mutual authentication and secrecy. Chapter 4 defines authentication as zero knowledge and applies the definition to an authentication protocol. Chapter 5 applies the security-protocol definitions in a standard setting before moving onto lightweight applications in Chapters 6 and 7. Finally, Chapter 8 summarises the thesis and indicates future work.

Chapter 2

Notation

This chapter describes the notation used in this thesis to reason about agent knowledge and behaviour in a distributed system. In order to reason rigorously we need to formalise requirements, specifications and implementation designs. Requirements will often describe "who knows what" because of our concentration on security, for which epistemic logic is used, although its semantics will be refined in the next chapter to take into account the inability to learn the results of infeasible computations. Designs are described as state transition systems, for which the Z notation is used, incorporating epistemic predicates where appropriate.

2.1 Predicate calculus

Propositional and predicate calculus are of such widespread use that they scarcely need to be introduced; on the other hand that means various notations exist. We use the notation summarised in Table 2.1.

$\neg$	Negation
$\wedge$	Conjunction
$\vee$	Disjunction
$\Rightarrow$	Implication
if a then b else c	Conditional
$\equiv$	Equivalence
$\forall$	For all
$\exists$	There exist
$Qx \cdot P$	Prenex normal form
$\mathbb{B}$	Booleans
$:=$	Equals by definition
$\vdash$	Theoremhood

Table 2.1: Logical notation.

Frequently we find that a property consists of a conjunction of sub-properties. So it is convenient to use the Z "stacking" convention which expresses conjunction as newline within parentheses thus

$$Bonnie \wedge Clyde \text{ is written as } \left(\begin{array}{c} Bonnie \\ Clyde \end{array} \right) .$$

The notation $p[e/x]$ means replacement of free variable x by expression e throughout formula p. In particular x and e may be tuples of the same length. For example, if $p(x, y) := x \wedge y$ then

$$p[Bonnie/x] = Bonnie \wedge y$$
$$p[Bonnie, Clyde/x, y] = Bonnie \wedge Clyde \ .$$

When we introduce Z (Section 2.4) we shall also use that notation when p is a schema having observable(s) x.

Duality of the quantifiers $\forall$ and $\exists$ means:

$$\forall\, x \cdot p(x) \;\equiv\; \neg\, \exists\, x \cdot \neg p(x)\ ,$$

which saves quoting laws for both quantifiers. We regard them as infinitary conjunction and disjunction respectively. Consequently we have the law

$$\forall\, x \cdot p(x) \wedge q(x) \;\equiv\; (\forall\, x \cdot p(x)) \wedge (\forall\, x \cdot q(x))$$

yet only

$$\exists\, x \cdot p(x) \wedge q(x) \;\Rightarrow\; (\exists\, x \cdot p(x)) \wedge (\exists\, x \cdot q(x))\ .$$

Such laws we use without reference.

Quantifications are typed:

$$\forall\, x : X \cdot p(x) \quad \text{and} \quad \exists\, x : X \cdot p(x)\ , \tag{2.1}$$

where $x : X$ denotes variable x to be of type X. When the type X is empty the former is true and the latter is false.

Free variables are bound by the use of universal and existential quantifiers. The observation above that quantifications are viewed as infinitary propositions is formalised by the n-point laws. The two-point law is:

$$\forall\, x : \mathbb{B} \cdot p(x) \quad \equiv \quad p[0/x] \wedge p[1/x]$$

and its dual.

The one-point law turns predicates into propositions for a variable $x : X$ with some value α:

$$\begin{aligned} & \forall\, x : X \cdot (x = \alpha) \Rightarrow p(x) \\ \equiv\ & p[\alpha/x] \\ \equiv\ & \exists\, x : X \cdot (x = \alpha) \wedge p(x), \end{aligned}$$

which of course is consistent with the observation after Formula (2.1) since $x = \alpha$ is false.

2.2 Epistemic logic

Epistemic, or knowledge, logic is a modal extension of propositional calculus (see Fagin *et al.* [26]). It is invaluable in the analysis of distributed information systems (see Fagin and Halpern [26]), but began in philosophy with the Greeks and was formalised by Hintikka [41]. In the 1960's knowledge axioms were defined. More recently reasoning about knowledge has had applications in diverse fields such as economics, linguistics, artificial intelligence and computer science (see Fagin *et al.* [26, 28]).

Since epistemic logic is a modal logic its semantics is a many-worlds Kripke semantics [53]. In the context for distributed systems that is reworked by Fagin *et al.* [29, 28]). In this thesis, as is generally the case, we need to reason about knowledge of a proposition which is true with high probability (similar reasoning is also used by *Fagin et al.* [26], Halpern *et al.* [39, 38], and Halpern, Moses and Tuttle [36]). We have avoided any situation in which an arbitrary loop contains a block satisfying a property which is almost true, because errors may accumulate resulting in a loop failing the property. For instance common knowledge is expressed as a limit (Definition (2.10)) which in practice we shall limit to depth two (see Section 2.2.1), obviating the problem.

2.2.1 Knowledge predicates and laws

The modality $K_i\phi$ means that agent i knows property ϕ, and its dual $\neg K_i \neg \phi$ can be thought of as agent i considers ϕ possible. Epistemic logic is distinguished from other belief logics by the ability to know only truths,

$$\text{if } \vdash K_i\ \phi \text{ then } \vdash \phi\ . \qquad \text{Law (2.2)}$$

Agents are considered rational, so they know all the theorems of propositional calculus and propositional inference:

$$\text{if } \vdash K_i\ (\phi \to \psi) \text{ and } \vdash K_i\ \phi \text{ then } \vdash K_i\psi\ . \qquad \text{Law (2.3)}$$

An infinite regress of identical modalities is avoided by the introspection laws:

$$\begin{aligned} &\text{if} \quad \vdash K_i\ \phi \quad \text{then} \quad \vdash K_i(K_i\phi)\ , \\ &\text{if} \quad \vdash \neg K_i\ \phi \quad \text{then} \quad \vdash K_i(\neg K_i\phi)\ . \end{aligned} \qquad \text{Law (2.4)}$$

In other words if i knows ϕ then it knows that it knows ϕ and so on.

Knowledge is conjunctive:

$$\text{if } \vdash K_i\ \phi \text{ and } \vdash K_i\ \psi \text{ then } \quad \vdash K_i(\phi \wedge \psi)\ . \qquad \text{Law (2.5)}$$

The converse follows by Law (2.3), since $\vdash (\phi \wedge \psi \Rightarrow \phi)$.

We apply epistemic logic frequently to the case of i knowing the value of a variable, so use abbreviation $K_i x$ when $x : X$ is a variable to mean there is some value t for which i knows $x = t$:

$$K_i x \ :=\ \exists\, t : X \cdot K_i(x = t)\ . \qquad \text{Definition (2.6)}$$

We extend that notation to finite sets V of variables, to mean i knows the value of each variable in the set:

$$K_i\ V \ :=\ \forall\, x : V \cdot K_i x\ . \qquad \text{Definition (2.7)}$$

Since V is finite the universal quantification is finite. If V is empty then the universal quantification is vacuous and so true.

If G is a finite set of agents then the modality $E_G\phi$ means that every agent in G knows ϕ:

$$E_G\ \phi \ :=\ \forall\, i : G \cdot K_i\ \phi\ . \qquad \text{Definition (2.8)}$$

Iteratively, $E_G^1\phi := E_G\phi$ and for $n \geq 1$:

$$E_G^{n+1}\ \phi \ :=\ \forall\, i : G \cdot \ K_i(E_G^n\ \phi)\ . \qquad \text{Definition (2.9)}$$

$E_G^n\,\phi$ means that every agent in G knows that every agent in G knows that ... that every agent in G knows that "ϕ is true" holds, where the phrase "every agent in G knows" appears in the sentence n times. This approximates common knowledge to a finite "depth" (see Halpern *et al.* [37]), where the modality $C_G\phi$ means that ϕ is common knowledge among agents in G.

$$C_G\ \phi \quad := \quad \forall\, n : \mathbb{N}^+ \cdot E_G^n\,\phi\ . \qquad \text{Definition (2.10)}$$

In other words, ϕ is true and every agent in G knows ϕ, every agent knows that every agent knows ϕ, and so on *ad infinitum*. This infinite conjunction is expressed as a fixed point of the E_G operator:

$$C_G\ \phi \quad = \quad E_G(\phi \wedge C_G\ \phi)\ .$$

Common knowledge is impossible to achieve in distributed systems where agents communicate by asynchronous sending and receiving of messages unless it is present initially.

Theorem 1. *(Attaining Common Knowledge, [37]).*
If common knowledge of a property does not initially exist then no finite number of communications in an asynchronous distributed system can attain it.

Indeed when agents send messages back and forth, no fixed number of acknowledgements to acknowledgements and so on suffices to reach agreement. An acknowledgement may get lost which a receiving agent cannot distinguish from the acknowledgement never being sent.

Acknowledgement can be seen as achieving "I know you have received a value", without which that information is unattainable. Knowledge is gained or lost by communication (see [14]):

If B receives ϕ then $K_B\phi$ and if ϕ came from A then $K_BK_A\phi$.
Conversely if at some stage of a computation $\neg K_A\,\phi$
and at a later stage $K_A\,\phi$ then
between those two stages A received a message. Law (2.11)

This can be used to approximate common knowledge of ϕ between A and B to depth 2.

Lemma 1 (Acknowledgement)**.**

1. *If B receives ϕ from A then $K_BK_A\ \phi$.*
2. *If B then acknowledges ϕ to A, then $K_AK_B\ \phi$.*

Proof.

1. This is simply a restatement of the first part of Law (2.11).
2. This applies Law (2.11) to the receipt by A of the acknowledgement from B. □

Soundness of the laws is established in the semantics previously referred to. A feature of this thesis is that we are able to reason entirely using laws without explicit use of semantics.

2.2.2 Application

In this section we demonstrate the use of Lemma 1. Formalisations of this argument appear throughout the thesis.

For example, when a sender transmits data packets to a receiver via a bounded channel, it may require the receiver to acknowledge each item before sending the next in order avoid data being lost when the channel is full. This is called the Stop-and-Wait protocol [85], which is too strict; the sliding-window protocol [85] is used in practice. But it illustrates the importance of acknowledgement.

After receiving an acknowledgement of the previous packet the sender knows that the receiver has received it, and can move to the next state in which it sends the next packet. Without receiving an acknowledgment the sender has no way of knowing that the packet has reached the receiver.

2.2.3 Using K: The Clever Princess

In this section we demonstrate the use of epistemic logic on a mini case ($n = 2$) of a popular example.

In a kingdom ruled by an evil king his beautiful and clever daughter has come of age. Her suitors come from far and wide to undergo the canonical trials for her hand: slaying dragons, rescuing damsels, *etc.* One suitor has the support of the king because he is well connected but unfortunately not very bright.

Finally the suitors have been reduced to a shortlist of two for the deciding trial which of course includes the king's favourite, but also a candidate who has won favour with the princess for his commitment to befriending dragons and his humility. After consulting with his advisors the king suggests to his daughter a protocol for the final trial, which he assures her will put his favourite at a disadvantage and so should be acceptable to her.

He suggests to her that his favourite stand in the centre of the hall facing the throne. Behind him stands the other candidate also facing the throne. So the king's favourite cannot see the other candidate but the other candidate can see him. The king proposes to tell the candidates that he is about to put a coronet on each, starting from the rear, which is either gold or silver, at least one of which is

gold. The first to deduce the colour of his own coronet and shout it out wins it and the princess's hand. He tells his daughter that he will in fact put a gold coronet on both.

Naturally the candidates are keen but honest, so shout out if and only if they identify their coronet. The king elaborates to his daughter that his own favourite would not see the other and so must be at a disadvantage. Does she welcome his suggested protocol?

The princess reflects for a minute, imagining the situation. The front candidate could reason that if his coronet were silver, then the other candidate would reason that his must be gold and shout out. When that does not happen the front candidate knows his assumption that his coronet is silver must be wrong, and so is able to identify it correctly as gold. Moreover the other candidate, seeing a gold coronet, is able to infer nothing about his own. So the king's favourite alone is able to win.

Used to dealing with her father, the clever princess replies that she welcomes his protocol but sees no reason to put one candidate at a 'disadvantage'. Why not arrange them facing each other? Since that is symmetrical each has an equal chance. Naturally she thinks to herself that the quicker has an advantage, by reasoning as she has just done.

Out manoeuvred, the king agrees. We leave the story's ending to the reader.

Reasoning. We formalise the princess's reasoning in epistemic logic. We must be careful to use global information only if it is available to a suitor. In this example, communication is synchronous: by word of mouth or by sight.

Consider first the king's proposal. When the king announces to the assembled suitors that both coronets are gold or silver and at least one is gold, that becomes common knowledge. Assume the suitors in the final trial are $i : [0, 2)$ (notation explained in Section 2.3.1) where 0 is the king's favourite. The variable $c_i : \{g, s\}$

represents the type of i's coronet, g for gold and s for silver. Due to the king's announcement:

$$C(\forall\, i : [0,2) \cdot c_i \in \{g, s\} \ \wedge\ \exists\, j : [0,2) \cdot c_j = g)\ . \tag{2.12}$$

In the king's suggested configuration each suitor knows the configuration, with 1 observing the coronet of 0:

$$C(K_1 c_0 \wedge \neg K_0 c_0 \wedge \neg K_1 c_1 \wedge \neg K_0 c_1)\ . \tag{2.13}$$

When a suitor identifies his coronet he shouts out and so that fact becomes common knowledge

$$\forall\, i : [0,2) \cdot K_i c_i \Rightarrow C(K_i c_i).$$

Now, what about the princess's argument? From (2.12) and (2.13) candidate 0 knows

$$K_0(c_0{=}s \ \Rightarrow\ K_1(c_1{=}g))\ .$$

But 1 has not identified his own coronet, since $c_1 = s$ and $c_1 = g$ are both consistent with the data. Thus 0 infers

$$K_0(c_0{=}g)$$

by contrapositive and Laws (2.2), (2.3).

In the configuration suggested by the princess (2.12) remains true but (2.13) is replaced by

$$C(K_1 c_0 \wedge \neg K_0 c_0 \wedge \neg K_1 c_1 \wedge K_0 c_1).$$

Now the same reasoning as above applies to the quicker of the two suitors.

The same reasoning applies to any shortlist. That case $n = 2$ corresponds to the inductive step for general n. The case $n = 50$ of the princess's configuration is used in Fagin *et al.*'s book [29] under the title *The Muddy Children*.

$\bot$	Undefined
$x :\in E$	Select x uniformly at random from a nonempty finite set E
$X \setminus \{a\}$	Remove from set X the element a
$\#$	Cardinality
$\oplus$	Overriding of relations
$P \gg Q$	Piping output from P as input to Q
dom, ran	Domain and range of a relation
seq X	Finite sequences from X
$\fatsemi$	Forward relational sequential composition
$\mathbb{N}, \mathbb{Z}, \mathbb{R}$	Natural numbers, integers, reals
$X \leftrightarrow Y$	Binary relations from X to Y
$X \to Y$	Total functions from X to Y
$X \pfun Y$	Partial functions from X to Y
$X \rightarrowtail Y$	Bijections between X and Y
$x \mapsto y$	Maplet $\{(x, y)\}$
$\varnothing$	Empty set
$\mathbb{P}$	Power set
$\supseteq$	Super set

Table 2.2: Expressions and symbols.

Why is it necessary for (2.12) to be common knowledge in the princess's configuration, when both suitors see that there is at least one gold coronet? The quicker suitor relies on the common knowledge in (2.12) to depth 2 which is stronger than the depth 1 provided by sight.

2.3 Set notation

We assume the notation in Table 2.2. The following notation is less common.

2.3.1 Intervals

An interval refers to either a finite set or list of consecutive natural numbers. As a set

$$[a, b) = \{n : \mathbb{Z} \mid a \leq n < b\} \ .$$

In particular the interval $[a, a)$ is empty.

2.3.2 Lists

Finite lists of type $\mathbb{D}$ are defined recursively to be empty or starting with an element of $\mathbb{D}$. Thus the type seq $\mathbb{D}$ of lists of type $\mathbb{D}$ is defined

$$ls ::= [\,] \mid d.ls\ ,$$

where $d : \mathbb{D}$. Of course $d.[\,]$ is written $[d]$ and the nonempty finite list

$$d_0.(d_1.\cdots(d_{n-1}.[\,])\cdots)$$

is written $[d_0, \cdots, d_{n-1}]$.

For lists xs and ys, $xs{+\!\!+}\, ys$ is the concatenation of xs and ys. The list xs is a prefix of list ys if $ys = xs +\!\!+ zs$ for some list zs. For a list xs, $\#xs$ gives its length.

2.3.3 Sequential composition

In this thesis relations are more important than functions and so it is relational composition that is important. If $P : A \leftrightarrow B$ and $Q : B \leftrightarrow C$ then their forward sequential composition

$$P \fatsemi Q : A \leftrightarrow C$$

arises by abstracting the intermediate state:

$$(a, c) \in P \fatsemi Q \ :=\ \exists\, b : B \cdot \begin{pmatrix} aPb \\ bQc \end{pmatrix}\ .$$

The order is opposite to that of functional composition, which is why the name 'forward' composition is often used.

2.4 State transition systems

We think of the information systems encountered in this thesis as discrete dynamical systems, evolving with time (see Kamkuemah [45]). We express each as a data type, having state, an initial state, and operations under which the type evolves. Each operation has a precondition, and takes input, delivers output, and updates state. We use the Z notation since it provides conventions to simplify the description of states, the description of operations, and their collection into a data type (see Spivey [81], Duke and Rose [23]). We now give a summary.

If s is the system state before an operation, its value after the operation (presuming it terminates) is written s'. An operation is specified by a predicate with free variables s, $in?$, s', $out!$. The notation ? and ! alert us to input and output of values, respectively.

The precondition of an operation is the weakest condition on state and input under which it is able to achieve a final state and output. If an operation Op changes state A by accepting input $in?$ and yielding output $out!$ subject to invariant P,

$$
\begin{array}{|l}
\hline
Op \\
a, a' : A \\
in?, out! \\
\hline
P(a, in?, out!, a') \\
\hline
\end{array}
$$

then its precondition holds at states before and input for which P holds:

$$
\begin{array}{|l}
\hline
\text{pre } Op \\
a : A \\
in? \\
\hline
\exists\, a' : A, \exists\, out! \cdot P(a, in?, out!, a') \\
\hline
\end{array}
$$

The Z notation is "tuned" for abstract description. So if a variable is not mentioned in the body of a schema but is declared then it is assumed to satisfy true, i.e., to take any value of its type. When expressing designs close to code, an implementation in

particular, that decisions results in lengthy descriptions. So Lamport's TLA, which is executable, chose to modify Z's convention by adopting a more programming approach: an unmentioned variable is assumed to be unchanged. Here we need to describe both abstract and low-level designs. So we use, for instance,

$$
\begin{array}{|l}
\hline
State \\
x : X \\
y : Y \\
\hline
P(x, y) \\
\hline
\end{array}
$$

as usual for abstract descriptions,

$$
\begin{array}{|l}
\hline
\Delta State \\
x, x' : X \\
y, y' : Y \\
\hline
P(x, y) \land P(x', y') \\
\hline
\end{array}
$$

But for more concrete descriptions with the same *State* but a context in which y is unchanged, we use $\Delta State(x)$ to mean only observable x is changed:

$$
\begin{array}{|l}
\hline
\Delta State(x) \\
\Delta State \\
\hline
y' = y \\
\hline
\end{array}
$$

Unmentioned observables in the Δ statement (y above) are unchanged, as in TLA [55] or in fact any code.

The notation Δ allows operations to be described as state-transition operations, requiring the state-before and state-after to satisfy the state invariant. This feature is particularly useful and supports the non-operational specification of operations. See for instance the operation *Push* in the *Stack* example (Figure 2.1), where the precondition holds iff the invariant holds afterwards, $\#s' \leq n$, which implies $\#s <$

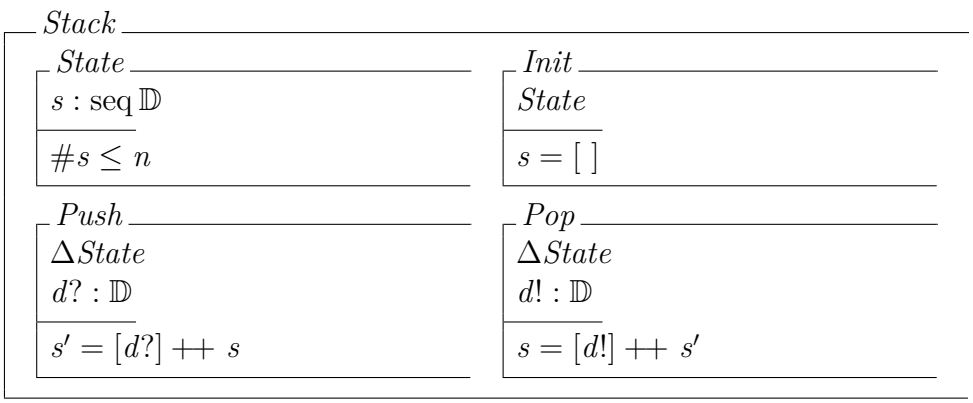

Figure 2.1: Bounded stack specified in Z. A *Push* adds input to the front of the stack and a *Pop* removes and outputs the front item. This example demonstrates the use of the implied state invariant, and the non-operational specification of *Pop*.

n. A more operational specification would require that to be included explicitly. This feature will abbreviate our specifications.

We consider a standard simple example of a bounded stack as a data type. See Figure 2.1. A stack stores at most n items and removes the most recent item first, where $n : \mathbb{N}^+$ is a generic parameter. Items are added with operation *Push* and removed with *Pop*.

Preconditions for stack operations are, liberated from schema form:

$$
\begin{aligned}
&\text{pre } \mathit{Push}(s, d?) \\
&= && \text{By definition} \\
&\exists\, s' : \mathit{State} \cdot \mathit{Push} \\
&= && \text{By body of schema } \mathit{Push} \\
&\exists\, s' : \mathit{State} \cdot s' = [d?] \mathbin{+\!\!+} s
\end{aligned}
$$

$$
\begin{array}{ll}
= & \text{One-point Law (2.2) and definition of } \mathit{State} \\
\#([d?] \mathbin{+\!\!+} s) \leq n & \\
= & \#([a] \mathbin{+\!\!+} as) = (\#as)+1 \\
\#s < n \ . &
\end{array}
$$

Similarly,

$$
\begin{array}{l}
\text{pre } \mathit{Pop}(s) \\
= \\
\#s > 0 \ .
\end{array}
$$

Pop is specified non-operationally. By comparison an operational description is closer to code (in Python):

$$
\begin{aligned}
d! &= s[0] \\
s' &= s[1 :] \ .
\end{aligned}
$$

The existence of an initial state ensures that a state specification is consistent. The calculation of each operation's precondition ensures it acts consistently. Identification of the precondition helps to confirm the correctness of the specification of the operation. Those reasons help to ensure that Z specifications are consistent and correct.

2.5 Message passing

In a distributed system agents interact pairwise by message passing. And the system exhibits three kinds of event: an internal action at an agent (an update to its state); a send action at the sender (containing the message and its recipient); a

receive action at the receiver (containing the message and its sender). The events at each agent are linearly ordered in time. Sometimes, for instance in simple security protocols, instead of operations in Z we use the more primitive message passing sequence diagram

$$\begin{array}{lcll} A & \rightarrow & B: & x \\ B & \rightarrow & A: & x+1\ , \end{array}$$

for A sending x to B and B acknowledging with $x+1$. If it is desired to include the calculation by B of $x+1$ we write

$$\begin{array}{lcll} A & \rightarrow & B: & x \\ & & B: & y := x+1 \\ B & \rightarrow & A: & y\ . \end{array}$$

In complicated cases the update to y is defined by schema.

2.6 Conclusion

This chapter has provided notation and laws to reason about knowledge of agents in distributed systems, and in particular how knowledge is gained. It has also promoted our view of an information system as a discrete dynamical systems expressed in Z.

In order to express communication between agents we have used message passing between agents with disjoint state spaces, rather than agents with overlapping state spaces. The result is simplified reasoning; for instance we need not be concerned with side effects.

In using Z we shall exploit implicit preconditions and nonoperational conditions, as exemplified in Figure 2.1. The result will be simpler and more elegant formalisations.

Chapter 3

Security model

This chapter introduces the security model for a lightweight distributed system and the minimum requirements for a protocol to be secure in this model. Protocol security is often described in terms of authentication, secrecy, integrity, non-repudiation, and availability. These properties are achieved by various cryptographic techniques (see "Handbook of Applied Cryptography" [50]). However, the current work is not an extensive study of all these techniques.

Instead, we give definitions for mutual authentication and secrecy in terms of epistemic logic. In subsequent chapters we apply the definitions to Lamport's One-Time Password Authentication Scheme, the Signal Protocol, and the Long Range Wide Area Network Protocol. But in this chapter we show how the techniques work on a benchmark example, that of the Needham-Schroeder protocol [67], and analyse the Diffie-Hellman key-agreement protocol [21] for use later. Weaknesses of these two protocols are highlighted and solutions suggested and analysed.

This preparation will be used in Chapters 6, and 7 to analyse protocols designed for lightweight distributed systems.

3.1 Distributed systems

We assume agents in distributed systems have disjoint state and therefore communicate by message passing which we assume to be asynchronous. We study security in the standard model of cryptographic security (see Bellare and Rogaway [8]): an adversary is restricted by the amount of time and resources available to compute certain cryptographic functions. These functions create powerful authentication and secrecy mechanisms.

3.1.1 Adversary model

A common model for security protocol analysis is the Dolev-Yao adversarial model [22]. It limits the power of an adversary by placing restrictions on what it can do. Like the Dolev-Yao model, we consider passive and active adversaries. A passive adversary like an eavesdropper, observes and remembers communications and tries to decipher messages. An active adversary is a passive one which also has complete control of a communication channel: it can intercept, delay, delete and insert messages between agents.

However, it is infeasible for both types of adversary to decipher messages without the correct decryption keys (unless they obtain encryption keys via other means like cryptanalysis or social engineering attacks). We emphasise that both types of adversary are bound by polynomial-time constraints, that is, do not have quantum computers, making certain problems infeasible.

3.1.2 Infeasibility

We assume that certain computations are infeasible. For lightweight distributed systems, it is neccesary to analyse computations on devices with limited resources. If the device can be hacked by computing certain feasible functions, the protocol is not secure.

Definition 1. *[Infeasibility] A computation (like computation of a value) is infea-*

sible if there exists no implementation in the standard model of computation more efficient than a global search of the solution space.

For example factorising large numbers is infeasible. Further examples will arise in the following.

3.1.3 Hashing

Messages shared between agents are usually encrypted and/or signed. Decrypting messages without the right key is infeasible due to cryptographic hash functions which form the core of encryption and signature schemes. In this thesis, the properties of hash functions are important in reasoning about security of a system.

Definition 2. *A hash function* $h : \{0,1\}^* \to \{0,1\}^q$ *accepts input of bitstring of arbitrary length and outputs a hash value of length* q*. It satisfies the following properties:*

1. h *is quick to compute but given output* $y : \operatorname{ran} h$ *it is infeasible to find* x *such that* $h(x) = y$ *(one-way property) (i.e., it requires an exhaustive search to find such* x*).*
2. h^{-1} *is discontinuous: given* $x : \{0,1\}^*$ *and* $h(x)$*, perturbing* $h(x)$ *slightly,* $y \approx h(x)$*, it is still infeasible to invert* y*.*
3. *In particular, although the range is in general smaller than the domain so* h *is not injective, given* $x : \{0,1\}^*$ *it is infeasible to find* $z \neq x$ *yet* $h(z) = h(x)$*.*

Notice that use of "hash function" in cryptography is different from its use in hash tables in Information Systems, though there is a vague similarity justifying the same name. Examples of hash functions in cryptography are provided by the family of Secure Hash Algorithms (SHA-128, SHA-256, ... [61]).

3.1.4 Public-Key Cryptography

We assume that each agent A has a public key $pk(A)$ and an inverse private key $pr(A)$. The notation $\{m\}_{pk(A)}$ (resp. $\{m\}_{pr(A)}$) will denote encryption of message m with agent A's public (resp. private key) key. So

$$\{\{m\}_{pk(A)}\}_{pr(A)} = m = \{\{m\}_{pr(A)}\}_{pk(A)} \ .$$

The notation $\{m\}_{pk(A)}$ corresponds to encryption of m for sending securely to A, and $\{m\}_{pr(A)}$ corresponds to signature of m by A. Applying the public key first gives encryption. Anyone can encrypt using $pk(A)$ which we assume to be common knowledge but only A has $pr(A)$. Applying $pr(A)$ first gives signature by A. Only A has $pr(A)$ for signing but anyone can verify the signature using $pk(A)$. A signature ensures a message has not been tampered with since its verifiable origin (message integrity).

Lemma 2. *A signs a message, m with its private key $pr(A)$, and sends both m and $\{m\}_{pr(A)}$ to B. Similarly, B signs a message with its private key $pr(B)$ and sends both the message and the signature to A:*

$$\begin{aligned} A \to B: &\quad m, \{m\}_{pr(A)} \\ B \to A: &\quad m, \{m\}_{pr(B)} \ . \end{aligned}$$

When B verifies A's signature, $K_B K_A\, m$. Similarly, when A verifies B's signature, $K_A K_B\, m$.

Proof Suppose a verification algorithm *Ver* exists. *Ver* inputs the sender's public key, the sender's signature and plaintext message and outputs a single bit indicating whether or not the signature is accepted i.e,

$$Ver(A, m, n) = 1 \ \equiv \ \{n\}_{pk(A)} = m \ .$$

When B receives n from A, it verifies that A sent m by checking that $Ver(A, m, n) = 1$. If so, B knows that A knows m, $K_B K_A\, m$ by Law (2.11). Otherwise verification fails. Similarly, when A receives n' from B. If $Ver(B, m, n') = 1$ then A knows that B knows m, $K_A K_B\, m$. □

3.2 Security properties

3.2.1 Authentication

When distributed agents need to communicate securely, communication protocols enable them to do so. Communication starts with a handshake to establish a secure connection then continues with message-exchange sessions. The idea is that only the two agents communicating should be able to read what is sent. To prevent an intruder from compromising a session, every session is protected with a new session key. We refer to this as end-to-end encryption (an idea patented by Zeidler [87]).

We now provide four progressively stronger conditions describing coordination between a pair of agents.

An initiator A starts a handshake with a responder B and they establish a shared secret ϕ when each knows ϕ:

$$\mathit{Shared}(A, B, \phi) \;:=\; \begin{pmatrix} K_A\ \phi \\ K_B\ \phi \end{pmatrix} . \qquad \text{Definition (3.1)}$$

When each agent also knows the other knows ϕ, we say ϕ is endorsed between them:

$$\mathit{Endorsed}(A, B, \phi) \;:=\; \begin{pmatrix} K_A K_B\ \phi \\ K_B K_A\ \phi \end{pmatrix} . \qquad \text{Definition (3.2)}$$

By Law (2.2) *Endorsed* implies *Shared*.

If the agents who share a secret are the only ones who know it, we say the secret is private:

$$\mathit{Private}(A, B, \phi) \;:=\; \begin{pmatrix} \mathit{Shared}(A, B, \phi) \\ C \neq A, B \Rightarrow \neg K_C\ \phi \end{pmatrix} . \qquad \text{Definition (3.3)}$$

Evidently *Private* strengthens *Shared*. In the next chapter we shall view the second conjunct as formalising that C has no knowledge of ϕ.

If furthermore the secret is endorsed, each agent is assured the other knows ϕ rather than an imposter, and we say the connection is authenticated:

$$\mathit{Authenticated}(A, B, \phi) \;:=\; \left(\begin{array}{l} \mathit{Endorsed}(A, B, \phi) \\ C \neq A, B \Rightarrow \neg K_C\ \phi \end{array} \right) . \qquad \text{Definition (3.4)}$$

Endorsed (and so *Authenticated*) does not achieve common knowledge of ϕ within the group $\{A, B\}$ but approximates it to depth 2 (see Section 2.2.1, Law (2.9)).

For example, a web browser and a web server authenticate each other when they receive acknowledgments for their identities, chosen cryptographic algorithms and nonces.

We analyse two benchmark handshake procedures – the Needham-Schroeder public-key authenticated protocol (NS) and the Diffie-Hellman key-agreement protocol (DH). DH establishes a secure connection over which agents compute a shared secret. Only agents involved in DH should know the shared secret, so that *Private* (3.3) holds.

The Needham-Schroeder-Lowe (NSL) protocol [56] locks out the possibility of an intruder with an authenticated connection. Agents are assured of each other's identity after authenticating a shared secret which only they know: *Authenticated* holds for NSL, even though the man-in-the-middle attack shows it fails for NS.

3.2.1.1 Needham-Schroeder public-key authentication protocol

With the advent of distributed systems arose the need for secure communication. The Needham-Schroeder was proposed to provide security by means of public-key cryptography. Recall from Chapter 1 that the protocol purports to authenticate agents by ensuring each agent knows the other knows a shared secret and that they are the only ones that know it, thereby achieving the property *Authenticated*. However a man-in-the-middle (MitM) attack demonstrated by Lowe [56] (see Section 3.2.1.2) negates the assumption that participating agents are the only ones that know the secret. This violates the second conjunct of *Authenticated* (3.4).

This section shows how NS achieves the property *Endorsed* but fails to achieve the property *Authenticated*. We use the knowledge predicates defined in Section 2.2.1 to reason about the MitM attack and about how NSL mitigates this attack. The results give us confidence in the appropriateness of our tools, which we use on new situations in the remainder of the thesis.

The NS public-key authentication protocol creates an authenticated connection between two agents using public key cryptography. Each party is equipped with a public/private key pair. Public keys are assumed to be common knowledge. Agents exchange identities and nonces[1] (all encrypted), and compute a shared secret ϕ that only they know.

$$
\begin{array}{lll}
NS(A,B,\phi) := & 1.\ A \to B: & \{id_A, n_A\}_{pk(B)} \\
& 2.\ B \to A: & \{n_A, n_B\}_{pk(A)} \\
& 3.\ A \to B: & \{n_B\}_{pk(B)} \qquad \textit{where } \phi := \{n_A, n_B\}\ .
\end{array}
$$

Figure 3.1: The Needham-Schroeder public-key authentication protocol. The values n_A, n_B are nonces discarded after one use (consisting of two communications). The shared secret consists of $\phi = \{n_A, n_B\}$.

Assume that both A and B follow NS and they each know it. In Step 1, A initialises a nonce n_A and initiates a handshake with B by sending it the message $\{id_A, n_A\}_{pk(B)}$. By Law (2.2), $K_B\ n_A$. The message is encrypted with B's public key. B receives, and decrypts the message with its private key.

In Step 2, B initialises its nonce n_B. Using Law (2.2), $K_B\ n_B$. According to Law (2.5): $K_B\ n_A$ and $K_B\ n_B$ then $K_B\ (n_A \wedge n_B)$. B sends to A acknowledgement for n_A along with its n_B. When A receives the message, it decrypts it and by Law (2.2), $K_A\ n_B$. By Law (2.5), $K_A\ (n_A \wedge n_B)$ With an acknowledgement of its nonce n_A,

[1] A nonce is used once and deleted.

by the 'Acknowledgement' Lemma 1, A knows that only B could have decrypted its previous message sent in Step 1. So $K_A K_B \phi$.

In Step 3, A sends to B an acknowledgement for n_B. B receives $\{n_B\}_{pk(B)}$, decrypts it and now knows that only A could have decrypted the previous message, $\{n_A, n_B\}_{pk(A)}$. Hence $NS \vdash \mathit{Endorsed}(A, B, \phi)$.

This purportedly achieves authentication between A and B

$$K_C\ \phi\ \Rightarrow\ C = A, B\ .$$

However Lowe [56] showed that NS is susceptible to a man-in-the-middle, MitM, attack that violates the premise that only A and B know ϕ. In a MitM attack, an intruder impersonating A (as $C(A)$) learns ϕ by establishing communication with B and convinces B it is communicating with A.

3.2.1.2 Man-in-the-middle attack on NS

Lowe [58] described four similar types of MitM attack on authentication protocols. They all involve an intruder impersonating one agent and starting a communication session with another. Here we analyse the simplest MitM attack on the NS protocol.

Agent A starts a handshake with what turns out to be a malicious agent C. C impersonates A, written $C(A)$, and starts a session with B. C forwards any messages it cannot decrypt. A MitM attack on NS, MitM_{NS}, is asymmetric: A communicates with C, sending messages encrypted with C's public key; whilst B replies to C, thinking it is communicating with A and sends messages encrypted with A's public key.

$$
\begin{array}{llcll}
1. & A & \rightarrow & C & : \{id_A, n_A\}_{pk(C)} \\
2. & C(A) & \rightarrow & B & : \{id_A, n_A\}_{pk(B)} \\
3. & B & \rightarrow & C(A) & : \{n_A, n_B\}_{pk(A)} \\
4. & C & \rightarrow & A & : \{n_A, n_B\}_{pk(A)} \\
5. & A & \rightarrow & C & : \{n_B\}_{pk(C)} \\
6. & C(A) & \rightarrow & B & : \{n_B\}_{pk(B)}
\end{array}
$$

Figure 3.2: A man-in-the-middle attack in which C communicates with A and impersonates A, $C(A)$ in communicating with B.

Theorem 2. *A man-in-the-middle attack on NS, $MitM_{NS}$, allows C to learn the private key $\phi = \{n_A, n_B\}$, violating the third conjunct of Authenticated:*

$$MitM_{NS} \vdash (C \neq A, B) \wedge K_C\, \phi \ .$$

Proof The proof focuses on what each agent knows after it receives a message. Let $\phi = \{n_A, n_B\}$.

In Step 1, after A initialises its nonce n_A, it initiates a handshake with C. It sends to C an encrypted message containing its identity, id_A, and its nonce, n_A. C receives and decrypts $\{id_A, n_A\}_{pk(C)}$, and learns id_A and n_A. Thus by Law (2.2), $K_C\,\{id_A, n_A\}$.

In Step 2, B receives $\{id_A, n_A\}_{pk(B)}$ from $C(A)$ and decrypts it. B now knows id_A and n_A, i.e., by Law (2.2), $K_B\{id_A, n_A\}$.

In Step 3, after B its nonce n_B, it responds to $C(A)$ with its nonce and an acknowledgement of n_A. This message is encrypted with A's public key. When C receives the message, it cannot decrypt it. In Step 4, C simply forwards the encrypted message to A. A decrypts the message and by Law (2.2) learns $K_A\,\{n_A, n_B\}$. Using the 'Acknowledgement' Lemma 1, A now knows that C knows ϕ, $K_A K_C\, \phi$.

In Step 5, A responds to C with an acknowledgement for n_B. C receives and decrypts the message, and learns n_B. Since C already knows n_A from Step 1, it now knows n_B, so $K_C\,\{n_A, n_B\}$ using Law (2.2). By the 'Acknowledgement' Lemma 1, C also now knows that A knows n_B, so $K_C K_A\,\{n_A, n_B\}$.

In Step 6, C forwards the acknowledgement for n_B to B. B decrypts the message, and by Lemma 1, B now knows that A knows n_B. So, $K_B K_A\,\phi$.

As a result of the $MitM_{NS}$ interaction

$$\begin{pmatrix} K_A K_C\,\phi \\ K_C K_A\ \phi \\ K_B K_A\,\phi \\ (C \neq A, B) \wedge K_C\,\phi \end{pmatrix}. \tag{3.5}$$

This result violates the third conjunct of $Authenticated(A, B, \phi)$. □

Lowe [56] mitigates $MitM_{NS}$ by including B's id in Step 2 of Figure 3.3, which results in $\neg(K_C K_A\ \phi)$.

$$\begin{array}{rcll} NSL(A, B, \phi) & := & 1.\ A \to B: & \{id_A, n_A\}_{pk(B)} \\ & & 2.\ B \to A: & \{id_B, n_B, n_A\}_{pk(A)} \\ & & 3.\ A \to B: & \{n_B\}_{pk(B)} \end{array}$$

Figure 3.3: Needham-Schroeder-Lowe (NSL), where $\phi := \{n_A, n_B\}$.

Theorem 3. *The NSL protocol meets the specification for Authenticated*(A, B, ϕ)*,*

$$NSL(A, B, \phi) \vdash Authenticated(A, B, \phi)$$

where $\phi = \{n_A, n_B\}$.

Proof The proof shows what agents know after receiving messages. A and B follow NSL as in Figure 3.3. In Step 1, $K_B\ n_A$ according to Law (2.2). In Step 2,

by receiving an acknowledge, $K_A K_B\ \phi$, according to the 'Acknowledgement' Lemma 1. Finally in Step 3, $K_A K_B\ \phi$, again following the 'Acknowledgement' Lemma 1. The protocol satisfies the first two conjuncts of *Authenticated* (3.4).

Now suppose an intruder C impersonates A (denoted $C(A)$) to B and tries to learn ϕ, and violates the third conjunct of *Authenticated*. The *MitM* attack takes place as follows. A starts a handshake with C, and C impersonates A and starts a handshake with B.

$$\begin{aligned} &1.\ A \to C &&: \{id_A, n_A\}_{pk(C)} \\ &2.\ C(A) \to B &&: \{id_A, n_A\}_{pk(B)} \\ &3.\ B \to C(A) &&: \{id_B, n_B, n_A\}_{pk(A)} \end{aligned}$$

In Step 1, C receives and decrypts the message from A. According to Law (2.2), $K_C\ n_A$. In Step 2, B receives and decrypts a message from someone impersonating A. B makes the following deductions: $K_B\ n_A$ by Law (2.2). In Step 3, C receives a message encrypted with A's public key. It cannot do anything with this message so it simply relays the message to A in Step 4. In Step 4, A receives an acknowledgement for n_A but from an agent whose id differs from C. So $K_A(id_C \neq id_B)$. Therefore A deduces that $\neg K_A K_C\ \phi$. At this point the *NSL* stops. The third conjunct still holds.

No other agent could know ϕ because messages are encrypted. □

3.2.2 Secrecy

Oftentimes when participants exchange messages they use the same message encryption key over and over again. If an attacker were to record messages and some time later obtain the encryption key, it could decrypt recorded messages as well as future messages. To ensure communications remain protected, every message must be encrypted with a fresh encryption key. Protocols use key exchange mechanisms like the Diffie-Hellman key exchange protocol [21] for creating ephemeral encryption keys. Since keys are ephemeral, they cannot be used to decrypt recorded messages.

In other words if an encryption key is compromised, it cannot be used to decrypt past or future messages.

To ensure secrecy of encryption keys we use infeasibility to guarantee an intruder cannot compute past or subsequent keys from a current encryption key in polynomial-time with a non-negligible success probability.

Informally, forward secrecy ensures that past messages remain protected; while future secrecy ensures that future messages remain protected. In cryptography hash-based *key derivation functions* (KDF) [50] are used to generate encryption keys. Key exchange protocols that use KDFs with hash properties given in Definition (2), can guarantee forward and future secrecy. (Chapter 6 Section 6.2.2.1 gives a definition of a KDF). The KDF inputs a current encryption key and some random values, and outputs a new key.

We now give formal definitions of forward and future secrecy.

3.2.2.1 Forward secrecy

When a current encryption key is compromised it is infeasible to derive a past encryption from it.

Definition 3. *Forward secrecy means that an attacker cannot infer a past encryption key $k_{t'}$ from a compromised current session key k_t:*

$$FoS(A, B, k_t) : = \ \forall X \neq A, B, \ K_X\, k_t \ \Rightarrow \forall\, t' < t \cdot \neg K_X\ k_{t'} \qquad \textit{Definition}\ (3.6)$$

where X is an intruder.

3.2.2.2 Future secrecy

An intruder that compromises a current encryption key can easily compute a future encryption key if it knows the formula by which it is calculated. A KDF removes this possibility by inputting random values along with a current session key, and outputting an encryption key that appears random to an intruder. When a current

session encryption key is compromised it is infeasible to derive the next key from it.

Definition 4. *Future secrecy means an intruder cannot infer a future session key $k_{t'}$ from a compromised current session key k_t:*

$$FuS(A, B, k_t) :=\ \forall X \neq A, B,\ K_X\, k_t\ \Rightarrow \forall\, t' > t \cdot \neg K_X\ k_{t'} \qquad \textit{Definition}\ (3.7)$$

where X is an intruder.

We combine the two notions of secrecy and express them as *FFSec*. Suppose an intruder $X \neq A, B$ that is perchance able to infer a session key k_t at time t in time domain $\mathbb{T}$. Then it is not able from that to compute any other session key $k_{t'}$, past or future. We write $\vdash K_X\, k_t$ to express that X infers key k_t. *FFSec* expresses that $K_X\, k_{t'}$ is not feasibly computable by adversary X from k_t.

$$FFSec(A, B, k_t) :=\ FoS(A, B, k_t)\ \wedge\ FuS(A, B, k_t)\ . \qquad \text{Definition}\ (3.8)$$

For any time $t : \mathbb{T}$ and any key function assigning k_t to any time t, we define:

$$FFSec(A, B) :=\ \forall\, t : \mathbb{T}, \forall\, k_t\ \cdot FFSec(A, B, k_t)\ . \qquad \text{Definition}\ (3.9)$$

3.2.2.3 Diffie-Hellman key agreement

Diffie and Hellman [21] had the idea of public-key cryptography but it was Rivest, Shamir and Adleman [71] who proved that it was feasible, implemented by what is now called the RSA protocol. It is well-known now that similar ideas were obtained for GCHQ by Ellis and Cocks [25].

Agents use DH to generate a shared secret key over an insecure channel. The shared secret is used to compute new session keys which are updated in subsequent sessions [62]. Groups used often include prime finite fields $\mathbb{F}_p^* = (\mathbb{Z}/p\mathbb{Z}^*)$, finite fields $\mathbb{F}_{p^n}^*$, and elliptic curves over finite fields $E(\mathbb{F}_p)$. DH uses a multiplicative group $\mathbb{Z}_q^*$ of prime order q, with a group generator g. The group has the advantageous property that exponentiation is quick while computing discrete logarithms is infeasible (Definition (1)). This is called the discrete logarithm problem.

Definition 5 (Discrete Logarithm Problem, DLP)**.** *Given a finite cyclic group G, a generator $g \in G$, and target $y \in G$, find $x \in G$ such that $y = g^x$.*

In terms of epistemic logic, because agents are polynomially bounded, we express DLP as follows: an attacker A that knows g and target g^x does not know x, i.e.,

$$\neg(K_A\,\{g, g^x\} \Rightarrow K_A\,x)\ . \qquad \textit{Definition}\ (3.10)$$

In practice an attacker must not have any better than a uniform chance of guessing the discrete logarithm. The chance of guessing it is non-negligible which we express as Property (3.10) to avoid reasoning with probabilistic semantics of epistemic logic.

With the group set to $\mathbb{Z}_q^*$, the values q and g are common knowledge between A and B. A randomly selects a secret key $a :\in \mathbb{Z}_q^*$ and computes g^a; similarly for B with b and g^b. A then sends g^a to B and B sends g^b to A. A calculates $(g^b)^a$ and B calculates $(g^a)^b$, which of course are equal and so constitute a shared secret (see Figure 3.4).

$$\begin{array}{rrll} DH(A,B,g;\ \phi)\ := & & A: & a :\in \mathbb{Z}_q^* \\ & & B: & b :\in \mathbb{Z}_q^* \\ & A \longrightarrow B: & & \{g^a\}_{pk(B)} \\ & B \longrightarrow A: & & \{g^b\}_{pk(A)} \\ & & A: & \phi := (g^b)^a \\ & & B: & \phi := (g^a)^b\ . \end{array}$$

Figure 3.4: The DH protocol. Agents A and B exchange values which they use to compute a shared secret ϕ.

The DLP for non-trivially sized numbers is infeasible (Definition eq1) as we now show.

Theorem 4. *The DH implementation satisfies* $Private(A, B, \phi)$*,*

$$DH(A, B, g, \phi) \vdash Private(A, B, \phi) \ .$$

Proof Assume both A and B follow DH and moreover they each know that, and they each know other's public keys. When B receives g^a from A, it computes $\phi = (g^a)^b$ so according to Law (2.2), B now knows ϕ, $K_B\phi$. Similarly, A receives g^b, computes $\phi = (g^b)^a$ and learns ϕ, by Law (2.2), $K_A\phi$. Thus $Shared(A, B, \phi)$ holds. It is infeasible for an eavesdropper that observes g^a and g^b to compute g^{ab} unless it knows either a or b. Thus $Private(A, B, \phi)$ holds. □

In fact Theorem 4 is best possible in the sense that $Private(A, B, \phi)$ cannot be replaced by $Endorsed(A, B, \phi)$ because A and B are not aware the other knows ϕ at this point.

3.2.2.4 Man-in-the-middle attack on DH

After violation of $Authenticated(A, B, \phi)$ by NS, due to $MitM_{NS}$, we ask: what of a MitM on DH?

We illustrate how a MitM cannot violate DH. Suppose an eavesdropper C intercepts communications g^a and g^b between A and B respectively.

$$\begin{array}{llllll} 1. & A & \rightarrow & C & : & g^a \\ 2. & C & \rightarrow & B & : & g^a \\ 3. & B & \rightarrow & C & : & g^b \\ 4. & C & \rightarrow & A & : & g^b \end{array}$$

Figure 3.5: A man-in-the-middle attack on DH, $MitM_{DH}$.

Theorem 5. *A man-in-the-middle attack on DH,* $MitM_{DH}$*, does not reveal private key* ϕ *to intruder* C *and so does not violate* $Private(A, B, \phi)$*,*

$$MitM_{DH} \vdash C \neq A, B \Rightarrow \neg K_C\phi \ .$$

Proof We want to show that an attacker C can never learn the secret $\phi = g^{ab}$ known to A and B. C observes g^a and g^b shared between A and B. A and B follow the protocol strictly, A knows a and no one else including C knows it, $K_A\ a$ and $\neg K_C\ a$. Similarly for B, $K_B\ b$ and $\neg K_C\ b$. It is infeasible for C to compute ϕ without knowing either a or b. So $\mathit{Private}(A, B, \phi)$ holds. □

Theorem 5 is best possible in the sense that $\mathit{Private}(A, B, \phi)$ cannot be replaced by $\mathit{Endorsed}(A, B, \phi)$ because A and B are not aware the other knows ϕ.

3.3 Conclusion

This chapter has provided a security model for use in the chapters to follow. The model assumes an attacker with full control over a communication channel but who is bound by the capabilities of a classical computer. We have given probabilistic epistemic definitions of mutual authentication and secrecy and have shown how an attacker can violate them using a man-in-the-middle attack, which incorporate infeasibility. An example, namely the Needham-Schroeder (NS) protocol, has been used to show how mutual authentication and secrecy can be achieved. NS turns out to be susceptible to a man-in-the-middle attack that violates the authentication requirement. We have analysed a patch proposed and demonstrated by Lowe [56], showing it achieves mutual authentication as well as the notions of secrecy defined in the chapter. Another example, namely the Diffie-Hellman key exchange protocol was analysed and shown to achieve both mutual authentication and secrecy by allowing agents to establish a share secret.

These benchmark examples give us confidence in the methodology for specification and reasoning to be used in this thesis.

Chapter 4

Zero Knowledge

In the previous chapter we have calibrated our method on two benchmark applications. Here we exploit it to analyse security of Lamport's One-Time Password Authentication Scheme for use in Chapter 7. We continue to use authentication (3.4) which incorporates zero knowledge on the part of an eavesdropper, and discuss more of the background of zero-knowledge protocols. We also use the commit-reveal framework, explained in terms of Blum's coin-tossing protocol, to structure our description of Lamport's scheme.

This chapter extends the work of [48].

4.1 Zero Knowledge

The idea of zero knowledge was incorporated as the second conjunct of the definitions of *Private* (3.3) and *Authenticated* (3.4). In summary an agent C is said to have zero knowledge of ϕ:

$$ZK(C,\phi) := \neg K_C\,\phi\ . \qquad \text{Definition (4.1)}$$

The term zero knowledge was coined by Goldwasser, Micali and Rackoff [34] in the context interactive theorem proving. An interactive proof typically involves two

agents, one called a prover which tries to convince another called a verifier of the truth of a statement. Such a proof is said to be zero knowledge when the only knowledge gained by the verifier is the truth of the statement. Zero-knowledge interactive proofs have been used to authenticate participants in identification protocols like the Feige-Fiat-Shamir Identification Scheme [30], the Quisquater Identification Scheme (GQ) [35], and the Schnorr Identification Scheme [75]. These protocols prove zero knowledge of discrete logarithms and also satisfy completeness and soundness requirements. Completeness ensures that an honest verifier always accepts a proof by an honest prover. Soundness ensures that an honest verifier rejects the proof of a dishonest prover.

In a zero-knowledge proof an agent called the prover P, tries to convince another agent called the verifier V, of the truth of a statement ϕ through a sequence of interactions without revealing any information about ϕ. As a result:

$$K_V K_P \phi \wedge K_P \neg K_V \phi$$

and so $ZK(V, \phi)$.

In the following section we introduce a design, the commit-reveal scheme, which achieves zero knowledge for B whilst it is making a decision but ends with both A and B knowing ϕ. This scheme will help to structure the description and verification of Lamport's scheme.

4.2 Commit-Reveal Scheme

The idea of 'escrow' is required in everyday interactions between untrusted agents and so has online equivalents. The structure is that of a 'commitment scheme' [11]. For now we see how it can be implemented using a hash function.

Several distributed protocols use the idea of an agent A committing to a value, an agent B making progress on that assumption, and then A revealing to B its committed value to complete a transaction. The committed value is initially hidden

from B but binding on A. Since the agents are untrusted it must be ensured that B is not able to infer any information about the committed value before it is revealed, and A is unable to change its value between commitment and revelation.

During the first interaction A commits to a value but keeps it hidden from B. The value is binding to A and cannot be changed during subsequent interactions. At the same time, B must not be able to infer any information about the committed value before it is revealed.

In committing to a value, A chooses $a : \mathbb{B}^*$ randomly (because it must be careful not to allow B any chance of predicting its choice, A would be reluctant to use a deterministic expression for a as discussed in Section 3.1.3) and sends a hash of a, $h(a)$, to B which B stores as variable b:

$$\begin{array}{lll} \textit{Commit} & A & : \quad a :\in \mathbb{B}^* \\ & A \to B & : \quad h(a) \\ & B & : \quad b := h(a) \end{array}$$

Since h is a hash function and a was random, it is infeasible for to B can glean anything from $h(a)$, except that A has committed to a value of a:

$$K_B K_A \, a \wedge K_A \, \neg K_B \, a \ .$$

The function h is now not common knowledge as it was in the previous chapter. However the hash system being used (like SHA, recall Section 3.1.3) is common knowledge as is its property of not having conjugates. In the context of h and a as above, a conjugate hash function h^* satisfies:

$$h^*(\neg a) = h(a) \ . \qquad \text{Definition} \quad (4.2)$$

When appropriate, A reveals its value by sending a and h to B. So now B must ensure that A hasn't changed the value a since committing to it, and so checks $h(a)$ against its stored value b (using Property 2 of Definition (2) of a hash function,

and the lack of conjugates). If they are equal B updates b to equal a and accepts it; but otherwise B rejects it.

$$\begin{array}{lllll} \mathit{Reveal} & A \to B & : & a, h & \\ & B & : & \textbf{if } b = h(a) & \textbf{then } b := a \fatsemi \mathit{Accept}(a) \\ & & & & \textbf{else } \mathit{Reject}(a) \end{array}$$

In other words if $\mathit{Accept}(a)$ then $K_A K_B\ a$. Since the hash system does not contain conjugates B knows that A cannot cheat if B guessed correctly by revealing instead $\neg a$ and h^*.

The commit-reveal interaction between A and B is summarised in Figure 4.1 which contains both a specification *AbsCR* and a protocol implementing it, *ImpCR*.

4.2.1 Correctness

Since the states of A (respectively B) are the same in both the abstract and concrete types, no data representation is required and we show that *ImpCR* refines *AbsCR* by showing the operations of the former are operational refinements of the latter: they have weaker preconditions and stronger postconditions.

The 'Commit-reveal correctness' Theorem 6 shows the implementation meets the specification.

Theorem 6 (Commit-reveal correctness.)**.** *The concrete type ImpCR refines the abstract AbsCR: each concrete operation refines the corresponding abstract operation.*

Proof The implementation *Commit* operation is total and

$$\begin{array}{ll} \mathit{ImpCR.Commit}(a, b) & \\ \Rightarrow & \text{Property 2 of Definition (2) of } h \\ a :\in \mathbb{B}^* \ \wedge \ b = h(a) & \end{array}$$

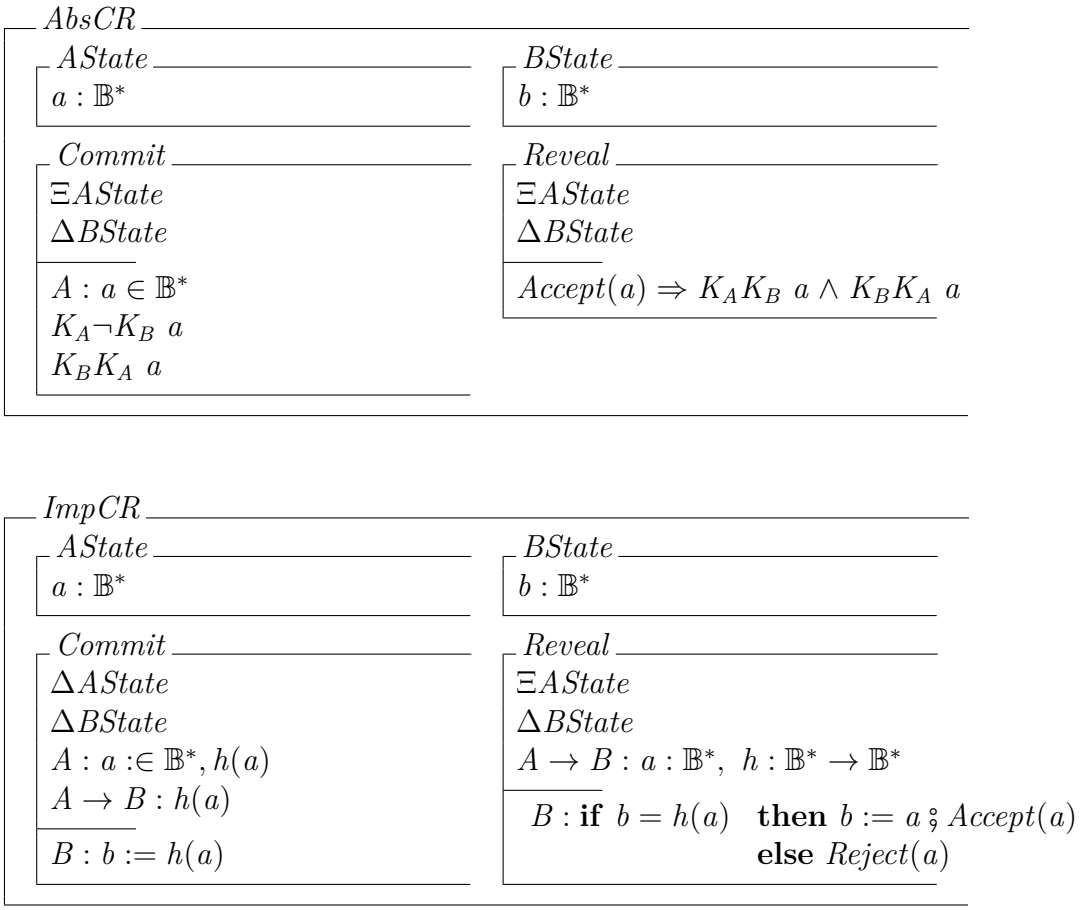

Figure 4.1: The commit-reveal abstract type *AbsCR* expresses its requirements in terms of knowledge. The implementation *ImpCR* gives a design. The *Commit* operation is specified to result in a pair of states about which A is assured its state is private from B, yet B knows A has committed to a value. The *Reveal* operation results in a state were the state of A is later revealed to B.

$\Rightarrow$ Law (2.2) and 'Acknowledgement' Lemma 1

$a \in \mathbb{B}^* \ \wedge \ K_B K_A \ a \ \wedge \ K_A \neg K_B a$

$=$ By definition

$AbsCR.Commit(a, b)$.

Restricting the implementation *Reveal* operation to the precondition $h(a) = b$,

$ImpCR.Reveal(a, b)$

$=$ Restriction

$b = h(a)$

$\Rightarrow$ By definition of the operation since no conjugates exist

$Accept(a) \ \wedge \ K_B K_A \ a \ \wedge \ K_A K_B \ a$

$=$ By definition (on precondition)

$AbsCR.Reveal(a, b)$.

□

4.2.2 Coin-tossing

An early use of the commit-reveal framework was due to Blum [10]. Suppose A and B want to flip a coin by telephone, a distributed situation with no trusted party. A chooses either 1 (heads) or 0 (tails). B wins if it guesses A's choice.

Let $a : \mathbb{B}$ be a variable which contains A's choice (either 1 for heads or 0 for tails). In committing to a value, A hashes its choice and sends to B the hashed value $h(a)$ – as commitment to its choice – then B sends to A its guess b. B must wait to receive A's commitment before sending its choice otherwise A would commit to the

opposite of B's choice. Now A knows who has won, but B does not. The commit step achieves zero knowledge: $ZK(B, a)$ holds.

It is important that h not be common because if it were B could simply compute $h(0)$ and $h(1)$ and know A's committed value of a.

In revealing a value, A sends to B its choice and the hash function, $A \to B : a, h$. B now also knows who has won, and can confirm that A did not cheat by checking that the hash function applied to A's guess gives the value A sent as its committed value (under the system assumption assumed above). That is,

$$\begin{array}{lll} \textbf{if } b = h(a) \textbf{ then } \mathit{Accept}(a)\text{;} & \textbf{if } b = a & \textbf{then } B \text{ wins} \\ & & \textbf{else } A \text{ wins} \\ \textbf{else } \mathit{Reject}(a)\ . & & \end{array}$$

By the 'Commit-reveal correctness' Theorem Blum's protocol achieves common knowledge to depth 2 of the fair winner.

4.3 Lamport's Scheme

Many devices connected to the Internet of Things use password authentication [43] but the problem with such an approach is that repeated use of the same password is vulnerable to attack [51]. The problem is further exacerbated when passwords are not frequently updated, which makes devices susceptible to more attacks and violations of device data. In Chapter 7 we shall use the following protocol to mitigate that problem.

One solution would be to allow a device user to derive a hashed value from its secret password each time it needs to be authenticated in a communication session. For instance, some transactions like online banking typically require a user to enter a one-time password before the user is granted access to the system. First use of the one-time password authenticates the user; any replay leads to failure. No information about the user's secret password is revealed but the bank is convinced the user knows the secret password.

$$
\begin{array}{rrl}
\mathit{Init} := & A: & n :\in \mathbb{N} \mid n \geq 2\,; \\
 & & a :\in \mathbb{B}^*\,; \\
 & & i := n-1\,; \\
 & & x := h^i(a)\,; \\
 & & y := h(x)\,; \\
 & A \to B: & y\,; \\
 & A: & i := i-1\,; \\
 & & X \\
X := & A: & \textbf{if } i \geq 1 \textbf{ then } A \to B: \quad x\,; \\
 & & \qquad\qquad\qquad\qquad\quad i := i-1 \\
 & & \qquad\qquad\quad \textbf{else } \mathit{Init} \\
 & B: & \textbf{if } h(x) = y \textbf{ then } B \to A: \mathit{Accept}\,; \\
 & & \qquad\qquad\qquad\qquad\quad y := x\,; \\
 & & \qquad\qquad\qquad\qquad\quad X \\
 & & \qquad\qquad\quad \textbf{else } B \to A: \mathit{Error}
\end{array}
$$

Figure 4.2: The L_AOTP Protocol, in which A reinitialises after n accesses, and uses a common hash function h.

Lamport's One-Time Password Authentication Scheme (L_AOTP) [54] was inspired by the commit-reveal framework to solve that problem.

Assume that A and B share the hash function h. A's state consists of an arbitrarily large natural n which equals the number of accesses it will perform, and a secret seed a. Initially A sends $h(a)$ to B which it stores. For its initial access, A computes $x := h^{n-1}(a)$ and sends it to B which confirms that h of that value equals its stored value $h^n(a)$. If so it grants A access, but not otherwise.

An eavesdropper who remembers $h^n(a)$ from the initial communication is unable to compute $h^{-1}(h^n(a))$, A's current password.

A and B iterate that process until A is unable to continue that descent, when it reinitialises n and a. See Figure 4.2.

The property satisfied by L_AOTP is summarised in the following theorem.

Theorem 7. *Assume that A, B are honest and follow L_AOTP. The protocol achieves Authenticated(A, B, x) on each iteration and, recalling future secrecy from Section 3.2.2.2, also $\forall\, i \cdot FuS(A, B, x_i)$ overall, where x_i denotes the value of x on the i-th iteration.*

Proof.

The X-iteration of the design of L_AOTP terminates because the variant i decrements on each iteration. Moreover it maintains the invariant

$$\begin{aligned} 0 \leq i < n \quad & \wedge \\ x = h^i(a) \quad & \wedge \\ y = h(x) \quad & \wedge \\ K_A K_B\, x \quad & \wedge \\ K_B K_A\, x \ . & \end{aligned}$$

The first three conjuncts follow by standard reasoning. The fourth conjunct follows since A receives *Accept* or *Error* from B on each iteration. The fifth conjunct follows since $A \rightarrow B : x$ on each iteration.

It remains to establish invariant $\forall\, C \neq A, B \cdot \neg K_C x$. On each iteration an eavesdropper C knows all values of x from previous iterations but is unable to compute the inverse $h^{-1}(x)$ of the latest x, which is the next value of x, because that is infeasible (Property 1 of Definition (2)). So $\neg\, K_C\, x$. □

Forward secrecy between A and B, $FoS(A, B)$, fails because, as just explained, given $h^i(a)$, an eavesdropper C can compute $h^{i+1}(a) = h(h^i(a))$, the previous key (*et seq.*).

4.3.1 Efficiency

We make a simplifying assumption for the purpose of approximating the amount of work a standard attacker needs to perform in order to break *ImpCR*. An attacker

will try to break *ImpCR* by inverting the scheme's hash function. We consider the Secure Hash Algorithm (SHA) with 256-bit key sizes (SHA-256). From Table 6.1, Chapter 6, we observe that a 256-bit key is associated with 128-bit security level. We let the attacker make $q \geq 2^{60}$ queries to the hash function while trying work out a one-time password. The attacker can compute the right one-time password with probability of success at most $q/2^k$, where $k = 256$. So the probability of computing the correct one-time password is approximately $2^{60}/2^{256} = 1/2^{196}$.

Now suppose a small IoT device with a 32MHz processor is used to implement Lamport's scheme. The clock ticks 32 million times per second. So with each tick, approximately one instruction completes. We place a lower bound on the number of computations an attacker can perform in order to work out a password: no less than $2^{128+60} = 2^{188}$ operations.

How much time would it take an attacker to find the correct one-time password? It would take $2^{188} \times \frac{1}{32000000} \approx 2^{188} \times \frac{1}{2^{25}} \approx 2^{163}$ seconds to compute the correct password, or roughly 1.104×10^{34} years.

4.4 Conclusion

As more resource-constrained devices connect to the internet, they introduce new security challenges. The need arises to protect these devices from attacks that compromise authentication. Threats indicate the need to analyse security guarantees and vulnerabilities associated with protocols envisaged for Internet of Things. One way to mitigate these threats is insist on zero knowledge by an intruder on secret values shared between participants in a protocol.

In this chapter we have defined the concept of zero knowledge epistemically and have illustrated its conceptualisation with a commit-reveal scheme. We have then shown how a protocol called the Lamport's One-Time Password Authentication Scheme achieves zero-knowledge authentication. A theoretical evaluation of the scheme has been provided which shows it is suitable for authenticating devices

with limited resources. This technique is to be exploited in Chapter 7.

Chapter 5

Vulnerability of TCP

Before moving to lightweight applications we investigate how our techniques apply in conventional communication networks. The purpose of this chapter is to formalise the handshake of TCP, the *Transmission Control Protocol* [70], which orchestrates communication, and so reason about it at a level of abstraction above any particular implementation.

TCP is used to exchange messages subject to the requirement that exchanges are protected. We assume TCP is implemented in a system that uses a public-key cryptosystem having public and private key pairs to protect data. The TCP procedure begins with a handshake phase to establish a connection that satisfies *Authenticated*, Definition (3.4), continues with the desired exchange, and ends by freeing up the connection. But if used in a non-standard manner, called a *split handshake*, the actions in the handshake phase constitute a known security vulnerability violating *Authenticated*.

5.1 Devices and connections

We consider communicating *devices* like computers, mobile phones, tablets, etc., gathered into connected networks. Networks may have firewalls for security and, to ensure privacy, different communications even between the same two devices occur

on different connections.

A 'connection' occurs between two endpoints, each called a *socket* and consisting of a port p within a device determined by its device identifier d. For that we suppose a set DID of device identifiers and a set $Port$ of ports common to all devices. We shall see later that ports are identified numerically.

Socket

$d : DID$
$p : Port$

A *connection* consists of: an identifier, *cid*; a pair of sockets, one for the *host* (which initiated the connection), the other for the *server* (which did not); a *mode* indicating which stage has been reached in the communication; and so-called *sequence numbers isn* "initial" and *jsn* (responding) for use by the communicating devices in acknowledging receipt of messages. Devices store that information which they pass to each other. However the firewall (as modelled here) needs only part of it.

We choose to express connection information in terms of a substructure C shared by connections, lookup tables and messages. It has an identifier *cid* in some set CID of connection identifiers, and origin and destination sockets.

C

$cid : CID$
$orig, dest : Socket$

Perhaps *orig* equals *dest*. So we impose no state invariant.

Then a connection consists of C, with *host* socket substituted for *orig* and *server* socket instead of *dest*, together with the sequence numbers and mode as already mentioned:

Connection

$C[host, server / orig, dest]$
$isn, jsn : \mathbb{N}$
$mode : open \mid synsent \mid \cdots \mid estab$

A *device* has: a device identifier, $i : DID$; a non-empty set P of ports which it uses for authenticated communications with other devices *via* sockets; and a set tcb (*transmission control block*, see the standard description [70]) of connections in which it is currently engaged. Each connection $c \in tcb$ has this device as either host or server[1] (i.e. the device identifier is either $c.host.d$ or $c.server.d$) and in either case the port is one of its own:

Device

$$
\begin{array}{l}
i : DID \\
P : \mathbb{P}\, Ports \\
tcb : \mathbb{P}\, Connection \\
\hline
P \neq \varnothing \\
\forall\, c : tcb \cdot \left(\begin{array}{l} (c.host.d = i) \lor (c.server.d = i) \\ c.host.d = i \;\Rightarrow\; c.host.p \in P \\ c.server.d = i \;\Rightarrow\; c.server.p \in P \end{array} \right)
\end{array}
$$

A message between devices contains a header and data. The header contains a connection (though with the original labels *orig* and *dest* of C) and a flag demarking the type of action being performed.

Header

$$
\begin{array}{l}
Connection[orig, dest / host, server] \\
flag : syn \mid ack \mid synack \mid rst \mid \ldots
\end{array}
$$

The data in a message, included for 'completeness', are of no concern in the handshake.

Message

$$
\begin{array}{l}
h : Header \\
data : \mathbb{D}
\end{array}
$$

[1] We see no reason to prohibit both.

5.2 The net

The *net* of connections under consideration forms a directed graph. We ignore the structure imposed by local area subnets, but do later take into account the effect of their firewalls. Our model is based on the description in Tenanbaum [85].

5.2.1 A directed graph

The net forms a *directed graph*[2] (S, E) whose *vertices* are sockets and whose *directed edges* consist of $c : C$ from $c.orig$ to $c.dest$ with label $c.cid$. It is assumed that edges are determined by their endpoints and also by their identifiers. Similarly devices are determined by their identifiers.

Net

$$S : \mathbb{P}\, Socket$$
$$E : \mathbb{P}\, C$$

$$\forall\, b, c : E \cdot \left(\begin{array}{l} b.orig = c.orig \;\Rightarrow\; c.orig \in S \\ b.dest = c.dest \;\Rightarrow\; c.dest \in S \end{array} \right) \equiv\; b.cid = c.cid$$
$$\forall\, e, f : \{g : Device \mid \exists\, s : S \cdot g.i = s.d\} \cdot e = f \;\Rightarrow\; e.i = f.i$$

In TCP, ports are allocated by service. For instance port 25 is assigned to email, port 80 to http, and those beyond 1024 to 'private conversations' (see Tenanbaum [85], Section 6.5.2).

Initially we assume that no private conversations are under way, but that public ports are accessible to all:

$$E \;=\; \{c : C \mid c.dest.p \in \{25, 80, \ldots, 1024\}\}\ . \tag{5.1}$$

Operations for the opening and closing of connections form part of TCP, as we shall see. The result will be a model of TCP as an abstract data type.

[2] Which could also be modelled as a multigraph whose vertices are devices.

5.2.2 Firewalls

Devices are grouped into local area networks, LANs, each protected by a firewall. Devices within a firewall communicate unimpeded. The exterior of a firewall is called its *DeMilitarised Zone*, DMZ. The DMZ contains publicly available servers which offer email, web, ftp and DNS services (see the text [86]). The DMZ is protected from the Internet by an external firewall. We consider simple external firewalls[3] which either allow or bar input on the basis of its origin and destination.

A firewall achieves that by having state consisting of a set, *safe*, of the IDs of devices it protects (i.e. inside the firewall) and for each such device a lookup table (set) of allowed connections. It stores only the part it needs, C, of a connection, not the full *Connection*. It is therefore convenient to use a function Γ which abstracts the extra values *isn*, *jsn* and *mode* of its argument, and returns the argument's values of *cid*, *orig*/*host* and *dest*/*server*. In terms of mathematics rather than Z, on the understanding that *Connection* has six observables and C has three,

$$\Gamma : Connection \to C$$
$$\Gamma(cid, host, server, isn, jsn, mode) = (cid, host, server)$$

We model a firewall as an abstract data type F, as usual, with state, initial state, an a filtering operation *OpF*.

The state of the firewall F is

StateF

$$safe : \mathbb{P}\, DID$$
$$lt : DID \pfun \mathbb{P}\, C$$
$$\mathrm{dom}\, lt = safe$$

Initially devices can communicate on their public ports, corresponding to Equation (5.1), but there are no private connections.

[3] See *Netfilter* [86] for a typical example of how much more firewalls can achieve.

InitF

$$StateF$$

$$lt = \{c : C \mid c.dest.p \in \{25, 80, \ldots, 1024\}\}$$

We overlook operations to add or remove devices inside the firewall, and concentrate on OpF, the filtering operation of F.

When the firewall receives an output message from a device it is protecting, it updates the component of its lookup table associated with that device by adding the C component of the message (i.e. Γ of the connection information in the message) before forwarding it. When both origin and destination lie within the firewall, it lets a message pass unimpeded (thus all internal destinations are included in a firewall's lookup table).

A message input from its DMZ is allowed to pass iff Γ of its connection lies (complete) in lt of its destination. Firewalls are assumed to be updated only on outputs. Several messages may be required to complete the firewall's information on a connection, as is the case in TCP's handshake. Whilst Γ of the connection information is being accumulated by the firewall, messages are delivered to a given port. A firewall does not support a private connection until its lookup table contains all the information in Γ of that connection. Using a conditional to express that:

OpF

$$\Delta StateF(lt)$$
$$m?, m! : Message$$

$$m?.h.orig.d \in safe \Rightarrow \left(\begin{array}{l} lt' = lt \oplus \{m?.h.orig.d \mapsto \Gamma(m?.h.connection)\} \\ m! = m? \end{array} \right)$$

$$m?.h.orig.d \notin safe \Rightarrow \left(\begin{array}{l} \text{if } \{\Gamma(m?.h.connection)\} \in lt(m?.h.dest.d) \\ \quad \text{then } m! = m? \text{ else } \bot \\ lt' = lt \end{array} \right)$$

Recall the convention that $\Delta StateF(lt)$ means that the observable *safe* of *StateF* remains unchanged (see Section 2.4).

If *Op* is an operation which outputs a message, then the effect of the firewall is described by piping the output of *Op* to the input of *OpF*. For then the output $m! : Message$ of *Op* is identified with the input $m? : Message$ of *OpF* and hidden, whilst the two operations *Op* and *OpF* act 'side by side'. The result is written using piping $Op \gg OpF$. Use of piping avoids an expansion of the result.

Having expressed a firewall in terms of state and a filtering operation, *OpF*, which acts on an output operation *Op* from within the firewall as

$$Op \gg OpF \ ,$$

we now investigate TCP's handshake.

5.3 Handshake

The purpose of this section is to specify TCP's handshake for establishing a connection that maintains TCP's privacy conditions (shown in the next section to be violated by the split handshake).

5.3.1 Specification

TCP is used to establish connections which are reliable (see Section 3.1.3 for message integrity) and authenticated (see Section 3.2.1 for the definition of authentication). The TCP handshake consists of a client and server of type *Device* (see Section 5.1), and operations *Syn*, *SynAck* and *Ack* (see Figure 5.1). The property we establish is the following.

Theorem 8. *As a result of TCP's handshake, connections are established satisfying:*

1. *each connection is characterised by its cid and moreover different connections have disjoint endpoints: for any connections b, c,*

$$\{b.host, b.server\} \cap \{c.host, c.server\} \neq \varnothing \;\Rightarrow\; b = c$$

2. *for host h and server s*

$$Authenticated(h, s, \phi) \ ,$$

where $\phi = \{cid, host, server, isn, jsn, mode\}$.

In this section we see how TCP's handshake achieves Theorem 8, and then in Section 5.4.1 how the split handshake violates it.

5.3.2 Opening a connection

The establishment of a connection, from a host device h to a device destined to be the server with socket ss, is begun by an operation *Open* being invoked at h. *Open* is supplied with a *port* and remote socket ss, and updates the device connection table tcb and the set P of ports. The operation proceeds with h updating its tcb to contain a new connection c having host device $c.host.d$ as $h.i$, host port $c.host.p$ supplied as *port*, connection server $c.server$ supplied as ss, an initial sequence number $c.isn$ chosen randomly and responder sequence number $c.jsn$ initially undefined (to be supplied by receiver's acknowledgement), and in mode $c.mode$ set to *open*, with a fresh host port, and with server having socket ss and

A port q of device h is *fresh* iff it is not used in h's tcb as either a host port or a server port:

$$Fresh(h, q) \;:=\; \forall\, b : h.tcb \cdot q \notin \{b.host.p, b.server.p\} \ .$$

The *Open* operation is expressed, from the previous paragraph,

Open

$$
\begin{array}{l}
h, h' : Device(P, tcb) \\
port : Port \\
ss : Socket \\
\hline
Fresh(h, port) \\
h'.P = h.P \cup \{port\} \\
\exists\, c : Connection \cdot \left(\begin{array}{l}
c.host.d = h.i \\
c.host.p = port \\
c.server = ss \\
c.isn :\in \mathbb{N} \\
c.jsn = \bot \\
c.mode = open \\
h'.tcb = h.tcb \cup \{c\}
\end{array} \right)
\end{array}
$$

We have extended the $\Delta S(x)$ notation by writing $h, h' : Device(P, tcb)$ to mean that the operation being specified does not change the third observable i of h.

Operation *Open* is total (that is, there are no preconditions) because there is always such a $c : Connection$ and a fresh port.

The operation *Listen* indicates a device s is waiting to receive incoming connection requests. *Listen* is supplied with a *port*. The operation checks that the *port* for listening on device s is a fresh one. Device state is updated with the new port number. A new connection is created with server device supplied as $s.i$, server port as *port*, initial sequence number *isn* chosen randomly and responder sequence number *jsn* initially undefined, and *mode* set to *listen*; the device *tcb* is updated with the new connection information. To handle multiple simultaneous connections through a single TCP port, many operating systems make a copy of the *tcb* and perform state transition and updates on the copy.

$$
\begin{array}{|l}
\textit{Listen} \\
\hline
s, s' : \textit{Device}(P, tcb) \\
\textit{port} : \textit{Port} \\
\hline
\textit{Fresh}(s, \textit{port}) \\
s'.P = s.P \cup \{\textit{port}\} \\
\exists\, c : \textit{Connection} \cdot \left(\begin{array}{l}
c.\textit{server}.d = s.i \\
c.\textit{server}.p = \textit{port} \\
c.\textit{host} = \bot \\
c.\textit{isn} :\in \mathbb{N} \\
c.\textit{jsn} = \bot \\
c.\textit{mode} = \textit{listen} \\
s'.tcb = s.tcb \cup \{c\}
\end{array} \right) \\
\hline
\end{array}
$$

Operation *Listen* is again total.

5.3.3 Operation *Syn*

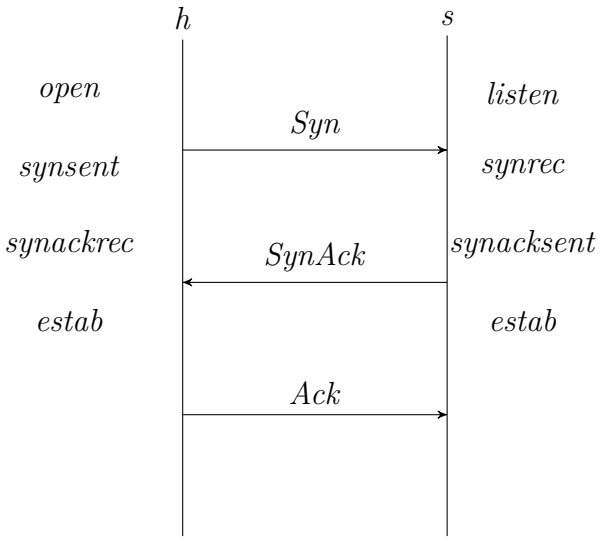

Figure 5.1: The 3-way TCP handshake. A host h and server s engage in a handshake that starts communication by 3 operations, *Syn*, *SynAck* and *Ack*. The vertical lines represent the states of h and s as time evolves downwards. After each operation, h and s result in different modes, *open*, *listen*, *synsent*, *synrec*, *synackrec*, *synacksent*, or *estab*.

The device h of *Open* starts an exchange of 3 operations, the handshake (see Figure 5.1), for each connection c in its *tcb* in mode *open*, to establish a connection with the (proposed) server of c. The first operation in the handshake is *Syn*, which we split into a send *SynSend* at the host and a receive *SynRec* at the server.

Operations are split in this way because it is convenient to express how the state of each device changes due to each suboperation.

Recall that operation *Open* updates the sender's *tcb* to contain connection c and sets the mode to *open*. Operation *SynSend* happens after that, ensured by its precondition including $mode = open$, and so it can use the information in c to define $m!$. It updates c's mode to *synsent* which is the only change to h's state.

The notation $c[synsent/mode]$ in schema *SunSend* means c's mode is updated with value *synsent*.

SynSend

$$
\begin{array}{l}
h, h' : Device(tcb) \\
ss? : Socket \\
m! : Message \\
\hline
\exists\, c : h.tcb \cdot \left(\begin{array}{l} c.host.d = h.i \\ c.host.p :\in P \\ c.server = ss? \\ c.jsn = \bot \\ c.mode = open \end{array} \right) \land \left(\begin{array}{l} m!.h.cid = c.cid \\ m!.h.orig = c.host \\ m!.h.dest = c.server \\ m!.h.P = c.host.p \\ m!.h.isn = c.isn \\ m!.h.jsn = c.jsn \\ m!.h.flag = syn \\ m!.data = \bot \end{array} \right) \\
\qquad \land \\
\qquad h'.tcb = h.tcb \cup \{c[synsent/mode]\}
\end{array}
$$

The precondition for *SynSend* is that there exists a connection in $h.tcb$ with host id equal to device $h.i$, host port is in P, and server socket equal to $ss?$, jsn undefined and $mode$ open. The connection is updated by changing only its mode.

pre *SynSend*

$$
\begin{array}{l}
h : Device(tcb) \\
ss? : Socket \\
\hline
\exists\, c : h.tcb, m! : Message \cdot \left(\begin{array}{l} c.host.d = h.i \\ c.host.p :\in P \\ c.server = ss? \\ c.jsn = \bot \\ c.mode = open \end{array} \right) \land \left(\begin{array}{l} m!.h.cid = c.cid \\ m!.h.orig = c.host \\ m!.h.dest = c.server \\ m!.h.P = c.host.p \\ m!.h.isn = c.isn \\ m!.h.jsn = c.jsn \\ m!.h.flag = syn \\ m!.data = \bot \end{array} \right)
\end{array}
$$

Message $m!$ is sent *via* h's firewall. The combination of *SynSend* and the firewall is expressed

$$SynSend \gg OpF \ .$$

Since $h \in safe$, the piping ensures that $m!$ is transmitted unchanged from its output by *SynSend* and lt is updated corresponding to the update of $h.tcb$.

Now h's firewall requires only the server's port in order to know both sockets of c and hence support a private conversation. That is achieved by operation *SynAckRec* (see Section 5.3.4).

Meanwhile message $m!$ is allowed through the receiver's firewall to port 25 of device $s = m!.h.dest$, which in operation *SynRec* inputs it as $m?$ and updates its tcb by updating an existing connection listening for incoming requests to one in *synrec* mode. There should be at least one listening connection on the server side.

SynRec

$$\begin{array}{l} s, s' : Device(tcb) \\ m? : Message \\ \hline m?.h.flag = syn \\ \exists\, c : s.tcb \cdot \left(\begin{array}{l} c.mode = listen \\ c.cid = m?.h.cid \\ c.host = m?.h.orig \\ c.server.d = m?.h.dest.d = s.i \\ Fresh(s, c.server.p) \\ c.isn = m?.h.isn \\ c.jsn :\in \mathbb{N} \\ s'.tcb = s.tcb \cup \{c[synrec/mode]\} \end{array} \right) \end{array}$$

The preconditon for *SynRec* is that the incoming message has *flag* equal to *syn*, and there exists a receiver connection in $s.tcb$ with connection *mode* equal to *listen* with host id equal to $m?.h.orig$, and destination id equal to server id.

$$
\begin{array}{|l}
\hline
\text{pre } SynRec \\
s : Device(tcb) \\
m? : Message \\
\hline
m?.h.flag = syn \\
\exists\, c : s.tcb \cdot \left(\begin{array}{l} c.mode = listen \\ c.cid = m?.h.cid \\ c.host = m?.h.orig \\ c.server.d = m?.h.dest.d = s.i \end{array} \right) \\
\hline
\end{array}
$$

As a result of *SynRec* the receiver knows the information in the message, known to the sender, as well as, of course, its own choices (of local port and *jsn*).

Lemma 3. *From operation SynRec, concerning its connection*

$$K_s\,(cid, host, server, isn, jsn, mode) \;\wedge\; K_h\,(cid, host, isn, mode)\ .$$

Proof We are assuming both h and s follow TCP (achieved in practice by the protocol being identified in the header) and moreover they each know that. Thus they know the operations invoked and their order. We also assume accurate delivery of messages.

So when s receives the message m from h it now knows the information therein, and remains (so far) unchanged. It stores and hence knows the values of *cid*, *host* and *isn*. By knowing TCP it knows that h has *synsent* for *mode*; and it stores the value *synrec*.

Also, since s chooses and stores *jsn* and its local port,

$$K_s\,(cid, host, server, isn, jsn, mode)\ .$$

This follows from the Knowledge Gain Law (2.11). □

5.3.4 Operation *SynAck*

We assume that the server wishes to continue with the connection.[4] It does so with a double operation, *SynAck*, whose decomposition in two we consider later in the 'split handshake'. Again we consider *SynAckSend* then *SynAckRec*.

In *SynAckSend* the server acknowledges the *Syn* operation concerning connection c by returning $isn := isn + 1$, its own sequence number jsn, and updating the mode of c to *synacksent*.

SynAckSend

$$
\begin{array}{l}
s, s' : Device(tcb) \\
m! : Message \\
\hline
\exists\, c : s.tcb \cdot \left(\begin{array}{l}
m!.h.cid = c.cid \\
m!.h.orig = s.i = c.server \\
m!.h.server.dest = c.host \\
m!.h.isn = c.isn + 1 \\
m!.h.jsn = c.jsn \\
m!.h.flag = synack \\
c.mode = synrec \\
s'.tcb = s.tcb \cup \{c[synacksent/mode]\}
\end{array} \right)
\end{array}
$$

The precondition for *SynAckSend* is that there exists a connection in $s.tcb$ with mode is equal to *synrec*, host socket equal to input *host*?, port in set P of ports, and server id equal to $s.i$.

[4] Otherwise the next operation would not be enabled by *SynRec* which would instead set *mode* to a new value.

pre *SynAckSend*

$$
\begin{array}{l}
s : Device(tcb) \\
host? : Socket \\
\hline
\exists\, c : s.tcb, m! : Message \cdot \left(\begin{array}{l} c.mode = synrec \\ c.host = host? \\ c.host.p :\in P \\ c.server = s.i \end{array} \right) \wedge \\
\qquad\qquad \left(\begin{array}{l} m!.h.cid = c.cid \\ m!.h.orig = s.i = c.server \\ m!.h.server.dest = c.host \\ m!.h.isn = c.isn + 1 \\ m!.h.jsn = c.jsn \\ m!.h.flag = synack \end{array} \right)
\end{array}
$$

The firewall of s incorporates information from $m!$ in its lookup table and forwards $m!$ to h. The result is

$$SynAckSend \gg OpF\ .$$

The firewall of the host $h = m?.h.dest$ allows the message through to h's port 25 and the following operation occurs. Operation *SynAckRec* inputs message $m?$ and updates an existing connection mode from *synacksent* to *synackrec*.

SynAckRec

$$
\begin{array}{l}
h, h' : Device(tcb) \\
m? : Message \\
\hline
m?.h.flag = synack \\
\qquad \exists\, c : h.tcb \cdot \left(\begin{array}{l} c.cid = m?.h.cid \\ c.server.p = m?.h.orig.p \\ c.isn + 1 = m?.h.isn \\ c.jsn = m?.h.jsn \\ h'.tcb = h.tcb \cup \{c[synackrec/mode]\} \end{array} \right)
\end{array}
$$

The precondition for *SynAckRec* is that incoming message $m?$ has flag equal to *synack*, and there exists connection with mode equal *synsent*, host id equal $h.i$ and server port equal to $m?.h.orig.p$.

pre *SynAckRec*

$$h : Device(tcb)$$

$$m?.h.flag = synack$$

$$\exists\, c : h.tcb \cdot \left(\begin{array}{l} c.cid = m?.h.cid \\ c.mode = synsent \\ c.host = h.i \\ c.server.p = m?.h.orig.p \end{array} \right)$$

Since h receives from s its original value *isn* incremented, it knows s has received the message containing it.

Lemma 4. *From operation SynAckRec, concerning its connection*

$$K_h\,((cid, host, server, isn, jsn, mode) \;\wedge\; K_s\,(cid, host, isn, mode))\ .$$

Proof Receipt by h of a message from s containing $isn + 1$ means (since delivery is assumed to be accurate) that h knows s received its original message and knows its contents. According to Lemma 1, upon receiving acknowledgement of *isn* from s, $K_h\, K_s\,(cid, host, isn, mode)$. The same reasoning, as in Lemma 3, applies.

From the message itself K_h completes its knowledge using the Knowledge Gain Law (2.11) so that $K_h\,(cid, host, server, isn, jsn, mode)$. □

5.3.5 Operation *Ack*

The final operation in the handshake, establishing as much common knowledge as is possible with just 3 operations, is *Ack*.

AckSend delivers a message from the host to the server, which requires few fields because a connection is uniquely determined by its *cid*. The host returns $jsn + 1$ to the server.

$$
\begin{array}{|l}
\textit{AckSend} \\
\hline
h, h' : \textit{Device}(tcb) \\
m! : \textit{Message} \\
\hline
\exists\, c : h.tcb \cdot \left(\begin{array}{l} m!.h.cid = c.cid \\ m!.h.jsn = c.jsn + 1 \\ m!.h.flag = ack \\ h'.tcb = h.tcb \cup \{c[estab/mode]\} \end{array} \right) \\
\hline
\end{array}
$$

The precondition of operation *AckSend* is that such a c exists in $h.tcb$ with mode equal to *synackrec*, server equal to $ss?$ and port number in set P.

$$
\begin{array}{|l}
\text{pre } \textit{AckSend} \\
\hline
h : \textit{Device}(tcb) \\
ss? : \textit{Socket} \\
\hline
\exists\, c : h.tcb, m! : \textit{Message} \cdot \left(\begin{array}{l} c.mode = synackrec \\ c.server = ss? \\ c.server.p :\in P \end{array} \right) \land \left(\begin{array}{l} m!.h.cid = c.cid \\ m!.orig = i = c.host \\ m!.dest = c.server \\ m!.h.jsn = c.jsn + 1 \\ m!.h.flag = ack \end{array} \right) \\
\hline
\end{array}
$$

The host's firewall is updated to contain the server's port,

$$\textit{AckSend} \gg \textit{OpF}\ ,$$

so from now on inputs are allowed by the firewall from private socket to private socket.

At the server the message passes through the firewall (it could use port 25 or be directed to the socket which the server has dedicated to this connection) and at the server updates the connection to *established* mode.

$$
\begin{array}{|l}
\hline
\textit{AckRec} \\
s, s' : \textit{Device}(tcb) \\
m? : \textit{Message} \\
\hline
m?.h.\textit{flag} = \textit{ack} \\
\exists\, c : s.tcb \cdot \left(\begin{array}{l} c.\textit{cid} = m?.h.\textit{cid} \\ h'.tcb = h.tcb \cup \{c[\textit{estab}/\textit{mode}]\} \end{array} \right) \\
\hline
\end{array}
$$

The precondition for *AckRec* is that the message $m?$ has flag equal to *ack* and incoming connection $m?.h.\textit{cid}$ already exists in $s.tcb$.

$$
\begin{array}{|l}
\hline
\text{pre } \textit{AckRec} \\
s : \textit{Device}(tcb) \\
m? : \textit{Message} \\
\hline
m?.h.\textit{flag} = \textit{ack} \\
\exists\, c : s.tcb \cdot \left(\begin{array}{l} c.\textit{cid} = m?.h.\textit{cid} \\ c.\textit{mode} = \textit{ack} \end{array} \right) \\
\hline
\end{array}
$$

Concerning the server's connection, using techniques that are now familiar,

Lemma 5. *Operation AckRec satisfies*

$$
\left(\begin{array}{l} K_s\, K_h\, \{\textit{cid}, \textit{host}, \textit{server}, \textit{isn}, \textit{jsn}, \textit{mode}\} \\ K_h\, K_s\, \{\textit{cid}, \textit{host}, \textit{server}, \textit{isn}, \textit{jsn}, \textit{mode}\} \end{array} \right) .
$$

Therefore, $\textit{Endorsed}(s, h, \phi)$, *where* $\phi := \{\textit{cid}, \textit{host}, \textit{server}, \textit{isn}, \textit{jsn}, \textit{mode}\}$.

□

Both devices have updated their firewalls to support an authenticated connection which we show establishes Theorem 8. An intruder i that wishes to obtain $\textit{cid}, \textit{isn}, \textit{jsn}$, needs to decrypt messages exchanged during *AckRec*. It can use either h's private key or s's private key. It can do so only if it is either h or s. That is,

$$
K_i\, \{\textit{cid}, \textit{host}, \textit{server}, \textit{isn}, \textit{jsn}, \textit{mode}\} \;\Rightarrow i = d, s\ .
$$

As a result of Lemmas 3, 4, and 5, the TCP handshake achieves authentication $Authenticated(h, s, \phi)$. This completes the proof of Theorem 8.

5.4 Split handshake

5.4.1 Specification

The 3-way TCP handshake described in Figure 5.1 is not the only way to create new connections. Some firewalls are configured to create TCP connections using a 4-way TCP handshake called a split handshake. In the 4-way handshake, the server splits the *SynAck* message into a separate acknowledgement *Ack* and synchronisation *SynAck*. There is no explicit instruction in the TCP documentation [70] on how to deal with a split handshake. A client following the TCP protocol is expected to silently accept the *Ack* and explicitly acknowledge the *Syn*. But this does not happen. TCP implementations [7, 66] that use the split handshake option produce unexpected behaviour. Figure 5.2 shows one effect of the split *SynAck*. When the server splits the *SynAck* into *Ack* and *Syn* messages, a client h establishes a connection, i.e., the mode is set to *estab*, when it receives the *Ack* message. Without full connection details and without following the protocol as documented, the server s manages to establish a connection. The client h then sends a *SynAck* to s. This reversal in the logical flow of the normal handshake causes h to behave like a server and violates the privacy of connections. This constitutes a known vulnerability (see [7, 66]). Some firewalls just drop the single *Ack* message from a server.

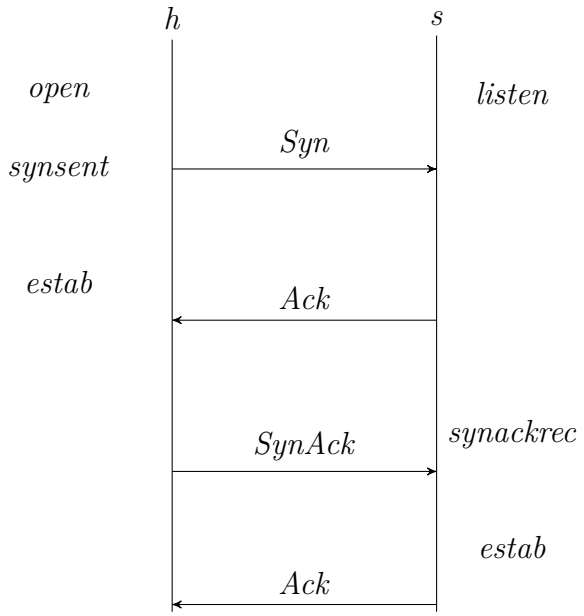

Figure 5.2: The 4-way split TCP handshake. A host h and server s engage in a handshake that starts communication by 4 operations, *Syn*, *Ack*, *SynAck*, and *Ack*. And after each operation, h and s result in different modes, *open*, *listen*, *synsent*, *synrec*, *synackrec*, *synacksent*, or *estab*.

The result of this section is:

Theorem 9. *Split handshake violates the invariant established by the TCP handshake that different connections have disjoint endpoints.*

So connection endpoints are now shared by connections with distinct *cid*s and therefore are no longer private.

5.4.2 Opening a new connection

This section introduces operation *SplitOpen* to formalise the idea of a split handshake. As before, a device h invokes *Open* which must start an exchange of 3 operations for each connection c in its *tcb* with mode *open* in order to establish a connection with the server of c. Again, *SynSend* chooses a c in $h.tcb$ with mode

open and updates c's mode to *synsent*. The same results hold as stated in Lemma 3.

The server s invokes operation *Ack* to acknowledge the connection request from the host.[5] $AckSend[h, h'/s, s']$ delivers *ack* from the server to the host, which returns $isn + 1$ to h. The host h silently[6] accepts the *ack* message with $AckRec[s, s'/h, h']$ before it times out while waiting for a *synack*. From operation *AckRec*, concerning its connection

$$K_h\, K_s\, (cid, host, server, isn, mode)\ .$$

The server then invokes operation *SplitOpen*, which works like *Open* but does not accept as input a socket and port of the device to connect to; it instead is supplied with an existing connection b. A new connection c is created with the same endpoints as connection b; and with mode *open* and a new initial sequence number *isn*. Connection c is then added to the *tcb* of s. The server goes on to create a connection request with *Syn*.

SplitOpen

$$s, s' : Device(tcb)$$
$$b : Connection$$

$$b \in s.tcb$$

$$\exists\, c : Connection \cdot \left(\begin{array}{l} c.host.d = s.i = b.server.d \\ c.host.p = b.server.p \\ c.server.d = b.host.d \\ c.server.p = \ \ b.host.p \\ c.cid :\in \mathbb{N} \neq b.cid \\ c.isn :\in \mathbb{N} \\ c.jsn = \bot \\ c.mode = open \\ s'.tcb = s.tcb \cup \{c\} \end{array} \right)$$

[5] This can be skipped if s goes directly to operation *Syn*.
[6] The *ack* field acknowledges *syn* sent.

5.4.3 Proof of Theorem 9

It remains to show that as a result of the *SplitOpen* operation, a new connection c is created from an existing connection b such that $c.cid \neq b.cid$ and yet

$$\{b.host, b.server\} = \{c.host, c.server\}. \tag{5.2}$$

In other words the invariant that connections are identified by both their ids and their endpoints is violated.

Assume both s and h follow the TCP protocol. The new connection c must have a different identifier from that of connection b due to *SplitOpen*:

$$c.cid \neq b.cid \ .$$

Yet also from *SplitOpen*:

$$\begin{pmatrix} c.host.d = b.server.d \\ c.host.p = b.server.p \\ c.server.d = b.host.d \\ c.server.p = \ b.host.p \end{pmatrix},$$

hence Equation (5.2) holds. □

5.4.4 Mitigating the split handshake

We consider the role the firewall has to play in the split handshake. Splitting *SynAck* in two causes unexpected behaviour. Instead of h merely acknowledging the *syn* from the server, it responds with a *synack* instead. The host firewall defined in the beginning lets these messages through and does not maintain state about connections. The problem with this is that the firewall will accept any connection request from a trusted server.

The host firewall must prevent unsolicited connection attempts before they reach a host. It does so with operation OpI, which inspects incoming messages. If the connection information is already in the lookup table, i.e., $\{\Gamma(m?.h)\} \in lt(m?.h.dest.d)$

and has the right flag, i.e, *synack*, *ack*, *rst*, the connection is updated in the lookup table lt. If the incoming message endpoints are not in lt and the message tries to access a well known port in E, the firewall lets the message through and updates lt. Otherwise it discards the message.

OpI is expressed as:

$$
\begin{array}{|l}
\hline
OpI \\
\Delta StateF(lt) \\
m?, m! : Message \\
\hline
\Gamma(m?.h) \in lt(m?.h.dest.d) \;\Rightarrow\; \left(\begin{array}{l} m?.h.flag \in \{synack, ack, rst\} \\ m! = m? \end{array} \right) \\
\Gamma(m?.h) \notin lt(m?.h.dest.d) \;\Rightarrow \\
\quad \left(\begin{array}{l} m?.h.dest.p \in E \;\Rightarrow \\ \quad \left(\begin{array}{l} m?.h.flag = syn \\ lt' = lt \;\oplus\; \{m?.h.dest.d \mapsto \Gamma(m?.h)\} \\ m! = m? \end{array} \right) \\ m?.h.dest.p \notin E \;\Rightarrow\; \left(\begin{array}{l} lt' = lt \\ m! = \bot \end{array} \right) \end{array} \right) \\
\hline
\end{array}
$$

So now the operation

$$OpI \gg SynRec$$

on host h drops the *syn* request when a port is private. It mitigates *SplitOpen* by ensuring that connections are determined by both their ids and their endpoints.

SplitOpen should not be confused with simultaneous open where ports are known before the connection set up. With *SplitOpen* the server lets the host expose its port number before it sends a *syn* request.

5.5 Conclusion

This chapter has formalised the TCP handshake and reasoned about an odd behaviour of the TCP handshake called a split handshake, that results in unspecified behaviour during operation. This treatment of the split handshake shows that formalising protocols can identify loopholes that would otherwise only show up when the protocol is implemented.

TCP underpins standard communications protocols and has been the first real use of our methodology. In the rest of the thesis we consider more recent protocols developed in response to lightweight and mobile devices.

Chapter 6

Analysis of Signal

So far this thesis has considered how our techniques work for a standard communications protocol, the Transmission Control Protocol. We now apply our techniques to a protocol whose design philosophy is maintaining privacy of user data. The Signal communication protocol allows communicating devices like smartphones to set up authenticated communication sessions using a key-agreement scheme (recall *Authenticated* in Definition (3.4) in Chapter 3), and thereafter encrypt each message exchanged between a sender and receiver with a new encryption key in order to maintain privacy (recall *Private* in Definition (3.3)) as well as secrecy of messages (recall *FFSec* in Definition (3.8)). This chapter formalises the protocol's key-agreement and encryption key-update mechanisms. It extends our paper [46] on analysing security in Signal.

6.1 Signal Protocol

The Signal Protocol [62] provides security for the Signal messaging application.[1] The messaging application encrypts data end-to-end, which means only the sender and receiver can read encrypted messages between them. Traditionally messaging servers are responsible for encrypting user data but with increasing public concern

[1]The protocol and app share the same name. Henceforth, by the name Signal we refer to the protocol.

about privacy of user data, protocols like Signal are sought to provide end-to-end encryption. A recent example of such concern was [74].

How does the Signal security protocol provide end-to-end encryption? The protocol sets up an authenticated connection for secure exchange of messages. Fresh symmetric keys encrypt and decrypt messages in a communication session and are updated in each session according to specifications by *Spike et al.* [60]. The protocol uses Diffie-Hellman key agreement to establish a shared secret key between sender and receiver. Session encryption keys are then derived from the shared secret key, and Signal's Double Ratcheting algorithm updates session keys for subsequent sessions. Fresh session keys ensure that past encrypted messages are secure against compromise of the current session key; as are future messages.

Cohn-Gordon *et al.* [17, 18] were amongst the first to formalise Signal. They concluded that Signal's key agreement and update mechanisms have no flaws under standard cryptographic assumptions. The authors also found that the protocol maintains forward secrecy.

Frosch *et al.* [31] formalised an early version of Signal called TextSecure. They showed that it achieves forward secrecy. The authors identified an authentication vulnerability where an intruder impersonates a legitimate user. They proposed a solution in which a participant proves knowledge of the private identity key before it is registered with a key-management server.

In contrast our formalisation of Signal uses epistemic logic to show that the protocol's key agreement and update mechanisms do achieve authentication and secrecy.

6.2 Key agreement

Signal agents use a sequence of sessions to exchange messages encrypted with symmetric keys. Keys are characterised as long-term, medium-term and prekeys which are used in the creation of symmetric encryption keys. A key-agreement algorithm

establishes a shared secret key between two communicating agents and then a key-evolution algorithm creates fresh symmetric keys to encrypt messages in subsequent sessions. These algorithms maintain forward and future secrecy.

Agents agree on a shared secret and use a key derivation function (KDF) to generate two sequences of session keys called chains – a sending chain and a receiving chain. One agent's sending chain matches the other agent's receiving chain. The sending chain contains message-encryption keys while the receiving chain contains decryption keys. The one-way nature of the KDF ratchets the chains forward so that output appears random to an intruder. Communication latency may temporarily force these chains out of synchronisation. The length of the chains is chosen to overcome this latency.

6.2.1 Initialising key agreement

Signal uses Diffie-Hellman key exchange (DH) (see Section 3.2.2.3) to create a shared secret key *ss* between two agents. Being shared, each agent knows the other knows it; being secret, they are the only ones who know it. That is formalised in terms of epistemic logic for agents A and B in Section 3.2.1 by predicate (from Definition (3.3)):

$$Private(A, B, \phi) \;=\; \begin{pmatrix} Shared(A, B, \phi) \\ C \neq A, B \Rightarrow \neg K_C\ \phi \end{pmatrix} . \tag{6.1}$$

Theorem 4 in Section 3.2.2.3 proves that DH satisfies *Private*.

6.2.1.1 Diffie-Hellman

Recall the definition of DH in Section 3.2.2.3. In the case of Signal, DH uses the group structure of a specific elliptic curve. We show in Section 6.5.1.1 the advantages of keys generated using this sort of curve.

As is illustrated in Figure 3.4, Section 3.2.2.3, the key values g^a and g^b can be observed by any intruder. However, an intruder will not be able to compute g^{ab} due to the infeasibility of solving the Discrete Logarithm Problem (see Definition (5), Section 3.2.2.3). DH as described in Figure 3.4 uses signed keys to verify the identity of agents involved in the key-agreement scheme. By using signed keys, the protocol satisfies *Authenticated*, Definition (3.4). We shall see in Section 6.5 how *Authenticated* is achieved with Double Ratcheting.

Signal calculates several shared secret keys by iterating DH. Section 6.2.2 shows how this is done.

6.2.2 Maintaining key agreement

New session keys are generated and updated in subsequent sessions with a KDF. The KDF inputs the shared secret key and other values, and outputs session keys. It is a one-way function that outputs keys of desired length. We use it in Signal's key agreement algorithm called Triple Extended Diffie-Hellman (X3DH). We now provide the definitions of a hash function and a key derivation function because they are used by the X3DH algorithm.

6.2.2.1 Key-derivation function

A key-derivation function inputs a key, a seed value, information about the protocol and a length argument indicating desired length of output. It outputs a key of desired length.

Signal uses a hash-based message authentication code [52] (HMAC) KDF. The HMAC is a keyed hash function.

Definition 6. *An HMAC function $f : \mathbb{B}^* \times \mathbb{B}^* \to \mathbb{B}^b$ inputs a key k in the form of a bit string, and a bit string of data, applies the hash function to those inputs, and outputs a hash value of length b:*

$$f(k, data) \quad = \quad h(k' \oplus u \mathbin{+\!\!+} h(k' \oplus v \mathbin{+\!\!+} data)) \ ,$$

where $++$ *denotes catenation of strings,* $\oplus$ *denotes addition modulo two,* u, v *are fixed bit strings of length* b *and*

$$k' = \begin{cases} h(k) & \text{for } \#k > b \ , \\ k & \text{otherwise} \ . \end{cases}$$

The output of f has the same length as the output of the hash function (see Definition (2)).

Recall the bijection between natural numbers and nonempty bit strings corresponding to 'binary representation', is defined recursively:

$$\begin{aligned} bin : \mathbb{B}^* \setminus \{[\,]\} &\rightarrow \mathbb{N} \\ bin\ [x] &= x \\ bin\ (xs ++ [x]) &= 2(bin\ xs) + x \ . \end{aligned}$$

Thus for instance $bin[1101] = 1.2^3 + 1.2^2 + 0.2^1 + 1 = 13$ in decimal notation (base 10). So the input string has most significant bits at the start.

Definition 7. *A KDF* F *inputs bit string key* k *and a bit string* r*, a bit string of context information* c *and a length* l*. It outputs a bit string of length* l

$$\begin{aligned} F : \mathbb{B}^* \times \mathbb{B}^* \times \mathbb{B}^* \times \mathbb{N} &\rightarrow \mathbb{B}^l \\ F(k, r, c, l) &= K_1 ++ \ \ldots ++ \ K_n \end{aligned}$$

where the right-hand side is evaluated using the HMAC function (typically SHA-256) f *to compute a key of size* l *and*

$$\begin{aligned} n &= \lceil l/b \rceil, \\ prk &= f(k, r), \\ K_0 &= [\,], \\ K_{i+1} &= f(prk, K_i ++ \ c ++ \ bin^{-1}(i+1)) \ , 1 \leq i < n \ . \end{aligned}$$

6.2.2.2 Extended Triple Diffie-Hellman

Signal uses three key types to create shared secrets: identity keys which are *long-term* keys that identify an agent; signed prekeys which are *short-term* keys that change periodically to provide authentication; and *one-time* prekeys which are used only once to provide forward secrecy.

Signal's key agreement protocol is called Triple Diffie-Hellman (X3DH) that computes at least three shared secret values (iterated n times, $n \geq 3$) by iterating DH. Extended Triple Diffie-Hellman key agreement can be represented as follows

$$XnDH(A, B, g, \mathbf{ss}) \ := \ \forall\, i : [1, n] \ \cdot \ DH(A, B, g, ss_i) \tag{6.2}$$

where $\mathbf{ss} = (ss_i)_{0 \leq i \leq n}$ is a vector of shared values.

Suppose that agents A and B want to communicate. Before sending an initial message to B, A obtains a vector $\mathbf{b}$ consisting of B's long-term public key ipk, signed public prekey spk, one-time public prekey opk and signature $sign$. After successfully verifying $sign$, A uses its private key ik and its private ephemeral key ek, and keys in $\mathbf{b}$ to compute shared secret keys and eventually an encryption key k_{enc}. See Figure 6.2 for how the keys are combined. In the meantime, A also creates vector $\mathbf{a}$ consisting of its public identity key ipk and public ephemeral key epk that corresponds with ek. A used ephemeral key is deleted hereafter. The message m is tagged with an authentication code ad and then it is encrypted using newly derived k_{enc}. A then calculates a tag ad using identities of both agents. A encrypts m and ad, $\{m, ad\}_{k_{enc}}$. Finally, A sends to B vectors $\mathbf{a}$ and $\mathbf{b}$ of public keys along with the encrypted message.

When B receives the message it retrieves $\mathbf{a}$ and $\mathbf{b}$, and loads its private keys corresponding to $\mathbf{b}$. It calculates shared secrets, and derives a decryption key. It also computes verification tag ad'. B decrypts $\{m, ad\}_B$ and checks that ad' matches ad. If verification succeeds, B deletes the one-time key opk and continues with the protocol.

A design for one iteration of $X3DH$ is shown in Figure 6.1.

$$
\begin{aligned}
A: \quad & \mathbf{a} = \begin{pmatrix} ik \\ ek \end{pmatrix} \\
A \longrightarrow S: \quad & A \text{ requests } B's \text{ keys} \\
S \longrightarrow A: \quad & \mathbf{b} = \begin{pmatrix} ipk \\ spk \\ opk \\ sign \end{pmatrix} \\
A: \quad & \mathbf{ss} = \begin{pmatrix} g^{ik \cdot spk} \\ g^{ek \cdot ipk} \\ g^{ek \cdot spk} \\ g^{ek \cdot opk} \end{pmatrix} \quad \text{⁘} \\
& m : \mathbb{D} \quad \text{⁘} \\
& k_{enc} = F(\mathbf{ss}) \quad \text{⁘} \\
& ad = h(ik +\!\!+ h(ipk)) \\
A \longrightarrow B: \quad & \mathbf{a}, \mathbf{b}, \{m, ad\}_{k_{enc}} \\
B: \quad & \mathbf{ss} = \begin{pmatrix} g^{ik \cdot spk} \\ g^{ek \cdot ipk} \\ g^{ek \cdot spk} \\ g^{ek \cdot opk} \end{pmatrix} \quad \text{⁘} \\
& k'_{enc} = F(\mathbf{ss}) \quad \text{⁘} \\
& \{\{m, ad\}_{k_{enc}}\}_{k'_{enc}} \quad \text{⁘} \\
& ad' = h(ik' +\!\!+ h(ipk')) \quad \text{⁘} \\
& ad' = ad
\end{aligned}
$$

Figure 6.1: X3DH (see Definition (6.2)) is designed to establish shared secrets, **ss**, and send initial messages encrypted with symmetric keys, k_{enc} and k'_{enc}.

X3DH supports the computation of shared secret keys using the pairs of exponents shown in Figure 6.2. The pair (ik, spk) authenticates agents. The pairs (ek, ipk), (ek, spk) and (ek, opk) ensure forward secrecy of keys. The concatenated shared secret keys are used with a KDF F to derive a master shared secret. The master shared secret is used with f again to derive initial symmetric session keys .

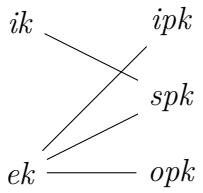

Figure 6.2: Shared secret key ss is computed from $g^{a_1 b_2}, g^{a_3 b_1}, g^{a_3 b_2}$ and $g^{a_3 b_3}$ as explained in Section 6.2.2. On the left are A's keys **a** and on the right are B's keys **b**.

6.3 Key evolution

Following X3DH, the key-evolution process updates session keys to ensure that past and future messages cannot be decrypted by knowing current session keys. Recall that session keys are changed in each session using a KDF. Signal's Double Ratcheting (DR) algorithm resets and updates session keys to achieve forward secrecy (recall Definition (3.6), Section 3.2.2). Since an intruder may know the KDF F and current session key, an extra precaution is required. Furthermore, session keys are temporary. So when a current session is compromised, past and future session keys will be infeasible to compute. Random values are introduced to randomize the input to the KDF so that an intruder cannot predict the output. This achieves future secrecy(recall Definition (3.7), Section 3.2.2).

Lemma 6. *Double Ratcheting satisfies forward secrecy and future secrecy: FFSec (recall Definition* (3.8)*).*

The rest of the section shows how Lemma 6 is achieved.

6.3.1 Session keys and chains

Recall that each agent maintains a sending chain and a receiving chain,[2] and the initiator's sending chain matches the responder's receiving chain and *vice versa*. The chains are initialised with a root key, which is the result of applying the KDF F to a DH key-agreement output. The process of initialising the root key, sending and receiving chains is called Diffie-Hellman ratcheting:

$$i = F(dh) \ , \tag{6.3}$$

$$s = F(i) \ , \tag{6.4}$$

$$r = F(s) \ . \tag{6.5}$$

Note that the KDF F accepts four inputs as in Definition (7). We include only the necessary inputs to highlight differences.

In Equation (6.3) the root key i is initialised from dh in Equation (6.3), the DH key-agreement output. The initial sending key s is initialised from the root key in Equation (6.4). The receiving key r is initialised from s in Equation (6.5). The new dh and KDF inputs ensure that the KDF output appears random using the third property in Definition (2). Random outputs ensure future secrecy.

Subsequent session keys of the receiving and sending chains are generated in a process called symmetric ratcheting as follows:

$$r' = F(r) \ ,$$

$$s' = F(s) \ .$$

Ratcheting forward maintains forward secrecy.

[2]We abstract the root chain and focus on root keys that initialise the sending and receiving chains.

6.4 Design

This section describes the state and operations for X3DH and Double Ratcheting. The design includes state for key information, messages, server and agents (that is, an initiator and responder). We also specify operations in X3DH and Double Ratcheting that change given state.

6.4.1 Key information

The type of data is written $\mathbb{D}$ as in Section 5.1. From it various data types are constructed. An important subset consists of identifiers:

$$Id : \mathbb{P}\,\mathbb{D}$$

So far we have used encryption and decryption by agents' private and public keys (see Section 3.1.3). Now we need notation for keys in abstract.

The type of keys is written $\mathbb{K}$. Participants in the protocol maintain a subset Pub of public keys and a subset Prv of private keys, which ideally partition $\mathbb{K}$, and a function $pair$ which provides a one-to-one correspondence between them.

$$\begin{array}{l} Pub, Prv : \mathbb{P}\,\mathbb{K} \\ pair : \mathbb{K} \leftrightarrow \mathbb{K} \\ \hline Pub \cap Prv = \varnothing \\ Pub \cup Prv = \mathbb{K} \\ pair \in Prv \rightarrowtail\!\!\!\!\!\twoheadrightarrow Pub \end{array}$$

A keypair has a private component, prv, and a public one, pub, which correspond.

KeyPair

$$\begin{array}{l} prv : Prv \\ pub : Pub \\ \hline (prv, pub) \in pair \end{array}$$

6.4.2 Key bundles

Various collections of keys are used. A key bundle consists of a long-term public key *ipk*, a signed public key *spk*, and a sequence *opk* (recall Section 6.2.2.2) of one-time public keys which turn out to be ephemeral. It also contains a string *sign* signed by the agent to identify it.

Bundle

$$ipk, spk : Pub$$
$$opk : \text{seq } Pub$$
$$sign : \mathbb{D}$$

A chain is a sequence of triples of keys, for which we introduce type X.

X

$$rk, ck, mk : \mathbb{K}$$

6.4.3 Messages

Messages in this design use a message type that consists of *Header* information and body information.

Message

$$h : Header$$
$$b : Body$$

The header includes the origin *orig* and destination *dest* of the message, a verification tag *ad*, and a *flag* that indicates the kind of message. For instance *req* is sent by an initiator to request a responder's key bundle; the flag *ack* is an acknowledgement of a *req* message; the flag *err* indicates an error message; and the flag *icomm* indicates an initial message in a communication. The header differs slightly from its use in Section 5.1. It would be unusual for the *orig* to be the *dest* but we do not prohibit it.

Header

$$
\begin{aligned}
&orig, dest : Id \\
&ad : \operatorname{seq} \mathbb{D} \\
&flag : req \mid ack \mid err \mid icomm \mid \ldots
\end{aligned}
$$

The message body consists of responder id *rid*, whose key bundle *bun* is being requested by initiator, *loc* which contains the initiator's public keys, connection information *Info*, and a sequence of *data*.

Body

$$
\begin{aligned}
&rid : Id \\
&bun : Bundle \\
&loc : Bundle \\
&info : Info \\
&data : \operatorname{seq} \mathbb{D}
\end{aligned}
$$

Info is shared between initiator and responder, and includes the conversation *cid* and *seed* value for use in the KDF (which is of course provided by the initiator).

Info

$$
\begin{aligned}
&cid : Id \\
&seed : \mathbb{N}
\end{aligned}
$$

6.4.4 Server

The server has its own id *sid* and a set *reg* of ids of registered participants for each of which it maintains a key bundle.

Server

$$
\begin{aligned}
&sid : Id \\
&reg : \mathbb{P}\, Id \\
&db : Id \pfun Bundle \\
&\hline
&\operatorname{dom} db = reg
\end{aligned}
$$

In this treatment we omit the routine operation of an agent registering its key bundle with the server, which is entirely straightforward.

6.4.5 Agent

The initiator and responder have their own ids and key information registered with the server. They also record the conversations in which they are involved. A conversation between an initiator *init* and a responder *resp* has:

- a unique id *cid*;
- a shared *secret*;
- double ratcheting values, root i, sending s and receiving r chain keys;
- a KDF function F, to initialise and update keys i, s, r respectively;
- the conversation can be at one of three stages, *IComm*, *Progress* or *End*.

The stage of a conversation is initialised by an interaction with the server, continues with an initial session (*IComm*) with a responder, matures to sessions (*Progress*) that exchange data and then ends (*End*).

Conversation

$init, resp : Id$
$cid : Id$
$secret : \mathbb{K}$
$i, s, r : X$
$F : \mathbb{B}^* \times \mathbb{B}^* \times \mathbb{B}^* \times \mathbb{N} \to \mathbb{K}$
$stage : IComm \mid Progress \mid End$

An agent has a unique *id*, identity key *idk*, signed key *spk*, a prekey bundle *opk*, its unique signature *sign*, *loc* which contains the initiator's public keys with a singleton list in *opk*, *rem* which associates responder id with corresponding key bundle, and a set *conv* of conversations in which it is engaged. Conversation ids are assumed to be unique, even for different conversations between the same two agents.

Agent

$$
\begin{array}{l}
id : Id \\
idk, spk : KeyPair \\
opk : \text{seq } KeyPair \\
sign : \mathbb{K} \\
loc : Bundle \\
rem : Id \rightarrow Bundle \\
conv : \mathbb{P}\, Conversation \\
\hline
\forall\, s, t : conv \cdot s \neq t \Rightarrow \ s.cid \neq t.cid
\end{array}
$$

6.4.6 Operation *KReq*

Having identified an agent with which it wishes to communicate (to become the responder) the initiator obtains the responder's key bundle from the server using operation *KReq*. (So communications are possible only between agents registered at the server.)

The operation *KReq* is split into a *Req* at the initiator, a receive *ReqAck* at the server, and *ReqAckRec* at the initiator. We describe those 3 operations separately rather than combining them, for simplicity.

Req inputs the responder $r?$ whose key bundle is to be requested from the server $s?$.

Req

$$
\begin{array}{l}
\Xi Agent \\
r?, s? : Id \\
m! : Message \\
\hline
m!.h.orig = id \\
m!.h.dest = s? \\
m!.h.flag = req \\
m!.b.rid = r?
\end{array}
$$

Message $m!$ is sent to the server.

In operation *ReqAck* the server acknowledges the initiator's message $m?$ and prepares a response message $m!$ containing part of the responder's key bundle. It returns the responder's bundle but with the *opk* chain replaced by just its head, and updates its state by deleting the head in the responder's *opk* chain. (Though natural to return the head value we model that by returning a singleton list containing the head, so that we can continue to use the type *Bundle*.) Thus the server updates *db* by replacing the responder *opk* sequence with the tail of *opk*. The server sets message $m!$ flag to *ack* and sends it.

Recall the notation (Section 2.4) $\Delta S(x)$.

ReqAck

$$\Delta Server(db)$$
$$m?, m! : Message$$

$$m?.h.dest = sid$$
$$m?.h.flag = req$$
$$m?.h.orig \in reg$$
$$m?.b.rid \in reg$$
$$m!.b.bun = db(m?.b.rid)[head(opk)/opk]$$
$$m!.h.orig = sid$$
$$m!.h.dest = m?.h.orig$$
$$m!.h.flag = ack$$
$$m!.b.rid = m?.b.rid$$
$$db' = db \oplus \{db(m?.b.rid)[tail(opk)/opk]\}$$

The precondition for *ReqAck* is that the message has flag *req*, is meant for the server and that the originator of the message is in *reg*. In reality that operation would be totalised by calling an error procedure outside the precondition, as usual.

pre *ReqAck*

$$
\begin{array}{l}
s : Server(db) \\
m? : Message \\
\hline
\exists\, db : s.db, m! : Message \cdot \left(\begin{array}{l} m?.h.flag = req \\ m?.b.rid \in \mathrm{dom}\ db \\ m?.h.orig \in reg \end{array} \right) \land \left(\begin{array}{l} m!.h.orig = sid \\ m!.h.dest = m?.h.orig \\ m!.h.flag = ack \\ m!.b.rid = m?.b.rid \end{array} \right)
\end{array}
$$

The message $m!$ is sent to the agent.

In operation *ReqAckRec* the initiator updates *rem* with the responder's key bundle.

ReqAckRec

$$
\begin{array}{l}
\Delta Agent(rem) \\
m? : Message \\
\hline
m?.h.dest = id \\
m?.h.flag = ack \\
rem' = rem \oplus \{m?.b.rid \mapsto m?.b.bun\}
\end{array}
$$

The precondition for *ReqAckRec* is that the message is meant for the initiator and forms an acknowledgement of its original message (as in the first two lines).

6.4.7 Initial communication operation

Having obtained the responder's key-bundle information from the server the initiator now commences communication with the responder.

The initiator creates a conversation with fresh *cid* which we describe using function

$$createConv : Id^2 \times \mathbb{K}^2 \times \mathbb{K}^3 \to Conversation\ .$$

It takes arguments x and y, the ids of the initiator and responder, and **a** and **b**, the vectors of keys in Figure 6.2.

$$\left(\begin{array}{ll} createConv(x, y, \mathbf{a}, \mathbf{b}).cid & :\in \mathbb{N} \\ createConv(x, y, \mathbf{a}, \mathbf{b}).init & = x \\ createConv(x, y, \mathbf{a}, \mathbf{b}).resp & = y \\ createConv(x, y, \mathbf{a}, \mathbf{b}).secret & = F(k, r1, c, l) \\ createConv(x, y, \mathbf{a}, \mathbf{b}).i & = F(secret, r2, c, l) \\ createConv(x, y, \mathbf{a}, \mathbf{b}).s & = F(i, r3, c, l) \\ createConv(x, y, \mathbf{a}, \mathbf{b}).r & = F(s, r4, c, l) \\ createConv(x, y, \mathbf{a}, \mathbf{b}).stage & = IComm \end{array}\right).$$

The final observable F in the schema *Conversation* in Section 6.4.5 is a KDF satisfying Definition (7) and *secret* is computed by the initiator from the vectors $\mathbf{a}$ and $\mathbf{b}$ as mentioned in Section 6.2.2.2:

$$k = F(bin^{-1}(g^{a_1 b_1}) +\!\!+\ bin^{-1}(g^{a_2 b_1}) +\!\!+\ bin^{-1}(g^{a_2 b_2}) +\!\!+\ bin^{-1}(g^{a_2 b_3}), r1, c, l) \quad (6.6)$$

where seed $r1$ (similarly for $r2$, $r3$ and $r4$), context c and length l of that description are as discussed in Definition (7).

In *IComm* the initiator updates its conversation set with the new conversation and sends an initial message $m!$ to responder $r?$. The message contains the initiator's key bundle *loc* and the responder's public key bundle.

IComm

$\Delta Agent(conv)$
$r? : Id$
$m! : Message$

$r? \in \operatorname{dom} rem$
$conv' = conv \cup \{createConv(id, r?, loc, rem(r?))\}$
$m!.h.orig = id$
$m!.h.dest = r?$
$m!.b.loc = loc$
$m!.b.flag = icomm$
$m!.b.bun = rem(r?)$

The precondition for *IComm* is that the initiator has the responder $r?$'s key bundle before the start of a conversation: the first line of schema *IComm*.

pre *IComm*

$$a : Agent(rem)$$
$$r? : Id$$

$$\exists\, c : a.rem, m! : Message \cdot \left(r? \in \operatorname{dom} c \right) \land \left(\begin{array}{l} m!.h.orig = a.id \\ m!.h.dest = r? \\ m!.b.loc = loc \\ m!.b.flag = icomm \\ m!.b.bun = rem(r?) \end{array} \right)$$

When the responder receives the initiator's public key-bundle and its own public key-bundle in message $m?$, it uses *JComm* to create a conversation matching the initiator conversation, deleting its one-time key used by the initiator. As before the responder uses private keys corresponding to entries in vector **b** and the public keys corresponding to entries in vector **a** to compute shared secrets.

JComm

$$\Delta Agent(opk, conv)$$
$$m? : Message$$

$$m?.h.flag = icomm$$
$$\exists\, is, js \cdot \left(\begin{array}{l} opk = is +\!\!+ [head(m?.b.bun.opk)] +\!\!+ js \\ opk' = is +\!\!+ js \end{array} \right)$$
$$conv' = conv \cup \{createConv(m?.h.orig, id, m?.b.loc, m?.b.bun)\}$$

The precondition for *JComm* is that the message has flag *icomm* and that the key bundle $head(m?.b.bun.opk)$ is in the responder's opk.

pre *JComm*

$$a : Agent(opk)$$
$$m? : Message$$

$$\exists\, o : a.opk, m! : Message \cdot \left(\begin{array}{l} m?.b.flag = icomm \\ m?.b.bin.opk \in o \end{array} \right)$$

6.4.8 Communication continued

A conversation between a given initiator and responder consists of an initial communication (*IComm* and *JComm*) followed by a finite number of double ratchet sessions. A conversation is defined:

$$Conv := IComm \fatsemi JComm \fatsemi DoubleRatchet^+ \ .$$

Recall each agent's conversation state includes variables for "double ratcheting":

- a seed dh to start the sequence of session keys for sending and receiving;
- a current sending key s and receiving key r.

We assume as given a Diffie-Hellman key exchange which contains an exchange and local updates resulting in the two agents sharing the secret key.

The operation which initialises the KDF F, but is also used any time to reinitialise secret keys (called DH ratcheting) is:

Init

$$\Delta Conversation(i, r, s)$$
$$dh? : \mathbb{N}$$

$$stage = IComm$$
$$i' = F(dh?, r1, c, l)$$
$$s' = F(i', r2, c, l)$$
$$r' = F(s', r3, c, l)$$

The precondition of *Init* that the conversation stage equals to *IComm*. In many situations *Init* is invoked on every new message exchange.

pre *Init*

$$conv : Conversation$$

$$\exists\, c, c! : conv \cdot \begin{pmatrix} c.stage = IComm \\ c! = c \end{pmatrix}$$

The operation *RatchetR* generates the next element of the receiving chain.

RatchetR

$$\Delta Conversation(r)$$

$$stage = Progress$$
$$r' = F(r, r4, c, l)$$

The precondition for *RatchetR* is that the conversation stage equals *Progress*.

Similarly *RatchetS*. Old keys i, r, s are deleted after they're updated. This maintains forward secrecy.

We have now formalised the idea of key exchange and key evolution via exchange of messages and session key updates.

6.5 Security analysis

We analyse the security of the Signal Protocol under the assumption of an intruder that is able to to intercept messages shared between an initiator and a responder (see Section 3.1.1). The goal of the intruder is to learn session-key information.

Theorem 10. *Signal is secure in the sense that it achieves Authenticated*(A, B, k) *of shared secret key* k *between initiator* A *and responder* B *who are not malicious and follow the protocol, and achieve FFSec*(A, B, k_t) *if* A *and* B *share secret key* k_t *at time* t*.*

Proof We show that operations involved in key agreement and key evolution achieve the security goals of authentication and secrecy in the presence of an intruder with Dolev-Yao capabilities.

After operations *IComm* and *JComm*, A and B establish shared secret key k using Equation (6.6). According to Lemma 2, successful verification of B's signature *sign* by A (again see Figure 6.1), means A knows that B knows k, i.e., *Endorsed*(A, B, k) (recall Definition (3.2)), and vice versa: B knows that A knows k, respectively. We consider two attack *scenarios*.

1. Suppose a passive intruder C observes communications between A and B. It cannot compute k due to the infeasibility of the discrete logarithm problem (DLP) embedded in Diffie-Hellman. Therefore, $\neg K_C\, k$.

2. Suppose an active intruder C compromises the key server and acts as a man-in-the-middle between A and B. We know from X3DH that agents are bound to a communication channel by their key bundles. The attacker may insert itself into the communication channel in the following ways:

   a) When A initiates communication with B, it instead receives the attacker's key bundle.

   b) When A and B are already in communication, the attacker shares new keys with both agents, forcing them to renegotiate session keys.

   The Signal protocol mitigates an attack by an active intruder by asking agents to verify identity keys during initial session setup (they do so by manually comparing a security code composed with A and B's identity keys) or warning agents when identity keys change. A man-in-the-middle does not go undetected unless both agents ignore verify request of identity keys (see Signal blog [59]). So, $C \neq A, B \Rightarrow \neg K_C\, k$.

Therefore, the communications *IComm* and *JComm* satisfy *Authenticated*,

$$Icomm, JComm \vdash Authenticated(A, B, k)\ .$$

Now we reason about forward and future secrecy and how key agreement and key evolution achieve these properties. A and B are non-malicious and follow the protocol, so do not reveal any session keys to show that for any $t : \mathbb{T}$ and for $C \neq A, B$, if $K_C\, k_t$ then $\forall\, t' \neq t \cdot \neg K_C\, k_{t'}$. We do so by cases.

For all cases, C observes **a** and **b** exchanged between A and B during the key-agreement phase.

If $t' < t$ then by the discrete nature of $\mathbb{T}$ and by Definition (3) (in Chapter 2), C inverts the KDF F in Equation (6.6) to obtain $k_{t'}$,

$$k_{t'} := F_{k_t}^{-1} \ . \tag{6.7}$$

Since A and B are non-malicious, they do not pass on information about keys to other agents. They engage in operations *RatchetR* and *RatchetS* to establish session keys. The only way for C to learn $k_{t'}$ is by computing it. Computing $k_{t'}$ is infeasible. Equation (6.7) cannot be computed even given $K_C\ k_t$. We infer $\neg K_C\ k_{t'}$ as required.

If $t' > t$ then this time

$$k_{t'} := F_{k_t} \ ,$$

where the inputs of function F are indistinguishable from random values to an intruder. Computing $k_{t'}$ without input r in Definition (7) is infeasible . Again, we infer $\neg K_C\ k_{t'}$ as required. Therefore, $\mathit{RatchetR}, \mathit{RatchetS} \vdash \mathit{FFSec}(A, B, k_t)$. □

Theorem 10 is best possible when computations are infeasible for an intruder who observes certain public values and not those that are private between non-malicious agents.

6.5.1 Computational cost

Signal uses elliptic-curve cryptography (ECC) for security. It uses the ECC *Curve*25519 equation developed by Bernstein [9] to generate keys. *Curve*25519 is fast and secure [9]. It also produces smaller keys with the same security strength as much larger keys produced by traditional public-key schemes like Rivest-Shamir-Adleman (RSA) [71] and Digital Signature Algorithm (DSA). This section presents the implementation results of a comparison of ECC with RSA and DSA.

6.5.1.1 Security level

The strength of a key is determined by its security level, which denotes the number of operations an attacker needs to perform to find a single bit of the key. Given a key of size q bits, a brute-force attempt to determine q bits performs 2^q search operations of the key space. The security level is usually set to $2^{q/2}$ (see recommendations by NIST [5]). Still, given a large enough q an exhaustive search of half the key space is infeasible in polynomial time (refer to Section 3.1.2).

What is feasible in polynomial time? The security level is based on the best known algorithms to compute a key. Computing a key in ECC is equivalent to solving the discrete-logarithm problem (see Chaum [15]). The best known algorithm for solving discrete logarithms is Pollard Rho (see Pollard [68]). It performs the following number of operations:

$$R(q) \quad = \quad \log_2(c\sqrt{2^q}) \ , \tag{6.8}$$

for some constant $c > 0$ and group order q. Its space requirement is proportional to $\sqrt{q}$.

RSA and other public-key signature schemes generate keys based on the product of two large primes. Security of these algorithms depends on the infeasibility of factoring this large product. The general number field sieve (GNFS) is the best known algorithm for factoring large integers (see Pomerance [69]). It performs the following number of operations:

$$L(q, v, u) \quad = \quad e^{(v+o(1))ln(q)^u ln(ln(q))^{(1-u)}} \ , \tag{6.9}$$

where constants v and u are determined by an optimised implementation. Its space requirement is proportional to $\sqrt{L(q, v, u)}$.

Equations (6.8) and (6.9) allow us in the next section to experiment using different key sizes in order to determine security levels for ECC and RSA key schemes.

6.5.1.2 Experiment 1

The National Institutes for Science and Technology (NIST)[5] recommends various security levels for different key lengths based on different key generation schemes. This experiment verifies these recommendations.

Equations (6.9) and (6.8) were tested for different key sizes and results are shown in Table 6.1. Results confirm NIST recommendations and show that smaller ECC keys have the same security level as bigger RSA keys.

Security level (bits)	ECC (bits)	RSA (bits)
80	160	1024
112	224	2048
128	256	3072
192	384	7680
256	521	15630

Table 6.1: Equivalence of ECC and RSA keys based on security level.

A 256-bit ECC key has the same security level or strength as a 3072-bit RSA key, while a 521-bit ECC key has the same security strength as a 15630-bit RSA key. Smaller keys suggest ECC is better suited for securing communications in distributed systems that have devices with limited processing power and memory.

Tables 6.2 and 6.3 show the results of GNFS and Pollard Rho used to approximate values in Table 6.1.

q	Scurity level	$L(q, \frac{64}{9}, \frac{1}{3})$
1024	80	86.76611925027707
2048	112	116.88381329581011
3072	128	138.73628085271660
7680	192	203.01873594416484
15630	256	269.38477262126889

Table 6.2: RSA with $L(q, \frac{64}{9}, \frac{1}{3})$.

q	Security level	$R(q)$
160	80	79.82537860389353
224	112	111.82537860389377
256	128	127.82537860389390
384	192	191.82537860389439
521	256	260.32537860389491

Table 6.3: ECC with $R(q) = \log_2(0.88\sqrt{2^q})$.

6.5.1.3 Experiment 2

We investigate the efficiency of RSA-based key generation and DSA signing algorithm. And then compare them to ECC algorithms, namely, NIST curve P-256 and Signal's *Curve*25519. P-256 and *Curve*25519 are both elliptic curves used for generating signatures (see Adalier [3]). We determine the average time to generate keys and then sign a 48-bit message. The experiment was run on an Intel Core i7 Generation laptop with 8GB RAM. We use Python NaCl and Crytodome libraries which contain implementations of these cryptographic algorithms. Table 6.4 shows the results.

DSA key generation is slower than RSA. DSA verification is also slower than RSA. However DSA signing is faster than RSA. Notably, ECC *Curve*25519 generates, signs and verifies much quicker than RSA-based algorithms. Again this suggests that ECC is better suited for securing communications in distributed systems where agents have limited resources.

(secs)	RSA (3072)	DSA (3072)	P-256 (256)	Curve25519 (256)
Key generation	4.2×10^{-1}	2.6	4.2×10^{-1}	4.7×10^{-5}
Verify (encrypt)	8.1×10^{-4}	1.7×10^{-3}	1.3×10^{-3}	8.0×10^{-5}
Sign (decrypt)	3.7×10^{-3}	1.0×10^{-3}	4.8×10^{-4}	3.1×10^{-5}

Table 6.4: Benchmark results obtained with Python NaCl and Cryptodome libraries. Comparing keys with the same security level.

Memory-wise, ECC keys are a better option than RSA keys. ECC keys are much smaller than the longer RSA keys with the same security strength. So smaller, stronger ECC keys could free up memory storing keys, and computational overhead associated with computing keys locally.

6.6 Conclusion

This chapter has reasoned about Signal security by using probabilistic epistemic logic to express its security properties, and by using the Z language to express state transitions for the Extended Triple Diffie-Hellman (X3DH) key agreement and Double Ratcheting algorithms. We have shown how the X3DH achieves authentication and establishes sessions keys. And also shown how the Double Ratcheting scheme updates these session keys and achieves forward and future secrecy.

Our methodology has shown that Signal is secure under the assumption of an intruder whose goal is to learn session keys. Additionally we have presented experimental results which show that elliptic-curve functions used in Signal generate session keys much quicker than standard cryptographic functions like RSA. The functions also generate smaller session keys with the same security strength as much larger keys generated by functions like RSA. These advantages make Signal suitable for implementation on devices with limited computing power.

Chapter 7

Vulnerability of LoRaWAN

We now apply our methodology to a lightweight security protocol. The purpose of this chapter is to formalise and analyse the Long Range Wide Area Network (LoRaWAN) protocol. According to the LoRaWAN documentation [4], the protocol is designed for Internet of Things (IoT) applications like sensor networks. It provides long-range communication for devices with limited resources while guaranteeing certain security properties. The LoRaWAN protocol starts with a handshake to establish an authenticated connection that satisfies *Authenticated*, Definition ((3.4)), continues with exchange of messages that are encrypted end-to-end. Unlike the Signal protocol, LoRaWAN does not encrypt each new message with a new encryption key. Static keys are vulnerable to misuse. A new key-management scheme is suggested to mitigate this vulnerability and thereby help to maintain secrecy of messages (recall *FFSec*, Definition (3.8)).

This chapter extends our paper [47] on a key-management solution for LoRaWAN.

7.1 Internet of Things

The IoT is made up of objects embedded with processing chips that autonomously connect to the internet and collect data. Innovations like automated mechanical systems, smart metering, remote monitoring, and autonomous-guided vehicles are

driven by networks of these devices. Typically, these devices have less processing power, less memory, and use security protocols designed to minimise power consumption. Bäumker *et al.* [6] showed how to minimise power consumption for typical LoRaWAN sensor devices.

Devices tend to optimise their limited resources for data processing at the expense of providing sufficient security guarantees. Security weaknesses arise like: devices trust the local area network they are in to such an extent that no further authentication is done in order to preserve power; or devices that are not password-protected or are protected with easy-to-guess default passwords in order to preserve memory. Because these devices are connected to the internet, these weaknesses can easily exploited by various attacks. A case in point is the Mirai attack that hijacked many unsecured IoT devices like DVRs and cameras and used them to send a massive 1.2 terabits per second to Dyn, a domain name lookup company. The attack left sites like PayPal, Twitter, Amazon and Netflix inaccessible for several hours (see "The Mirai botnet explained" [32]).

The rest of this chapter uses our approach to specify and reason about the behaviour of LoRaWAN and the security properties it achieves. It also highlights a known key-management vulnerability in LoRaWAN and suggests a new way to fix it.

7.2 LoRaWAN

Battery-powered end-devices like sensors collect data (e.g., the temperature in a building) and use the LoRaWAN protocol to transfer the data via gateways to remote servers for processing. Then devices go into sleep mode when they are not transmitting or receiving data. The devices, gateways and remote servers form a LoRaWAN network.

End devices start sessions with remote servers over which they transmit encrypted data. The devices and remote servers have identity keys called root keys, and use them to derive session encryption keys.

Before transmitting messages, a device requests to join the LoRaWAN network. The device initiates a handshake with a remote server in order to establish an authenticated session. The session is established using one of two handshake procedures: Over-The-Air Activation (OTAA) handshake; or Activation By Personalisation (ABP) handshake.

In an OTAA handshake, the end device sends a join-request message to the network server, and waits for a response called a join-accept message. If the network server approves the join request, it assigns the device a network address. The device and server then use their root keys to derive shared session keys.

On the other hand, end devices using the ABP handshake are preloaded with root keys, session keys and network addresses so there is no need to exchange join-request and join-accept messages in order to derive session keys.

Both handshakes have security issues. They both end up using static session keys to encrypt data. Updating these keys requires manual intervention like resetting a device. If a device is physically captured or cloned, an intruder can learn root keys and start computing session keys. One way to mitigate this vulnerability is to frequently introduce fresh values in the key derivation process. These values can then be used to update session keys. There is no clear indication in the LoRaWAN specification [1] how frequently session keys are updated. The lack of an update protocol for static session keys constitutes a vulnerability.

Selander *et al.* [76, 73] studied this vulnerability and proposed a key update mechanism that uses the Diffie-Hellman key-exchange scheme to create new keys after the OTAA handshake. In this work we propose updating session keys using a scheme similar to Lamport's One-Time Password Authentication Scheme (see Chapter 4).

7.3 The network

The LoRaWAN network consists of end devices and gateways which communicate with a core network. The core network consists of three servers, often co-located, that perform different tasks: the network server routes messages through the network; a join server handles join requests; and an application server processes data from end-devices. We abstract details of the core network, and model it as a single server.

The device and server each have a unique identifier, a network address, and nonce values. The values are drawn from the subsets,

$$Id, Addr, Nonce : \mathbb{D}$$

As before, the type of keys is denoted $\mathbb{K}$. Devices are assigned unique keys at manufacture, called root keys. We call them *Root* and they consist of an application key *appKey* for securing data, and a network key *nwkKey* for routing purposes.

Root

$$nwkKey, appKey : \mathbb{K}$$

During the handshake devices derive session keys, which we call *Sess*, from the root keys. The device-specific network session key, *nwkSKey*, is derived from *nwkKey*, and an application-specific session key *appSKey*, is derived from *appKey*, which is used by both the server and end device to encrypt messages.

$$Sess := Root[nwkSKey, appSKey / nwkKey, appKey] \ .$$

Messages exchanged consist of a header $hdr : Header$, payload *pyl* and message integrity check bit *mic*. The message is similar that in Section 6.4.3.

Message

$$hdr : Header$$
$$pyl : Request \mid Accept$$
$$mic : \mathbb{D}$$

We omit other details and include in the only header the ids of *orig* and *dest*.

Header

$orig, dest : Id$

The message payload indicates the type of message, whether it is a join request or join accept. The join request contains a join-server id *jsid* to handle join requests, a device id *did* and device nonce *devNonce*.

Request

$jsid, did : Id$
$devNonce : Nonce$

The join accept contains the join-server nonce *joinNonce*, the server network address *svrAddr*, and a device network address *devAddr* assigned by the server.

Accept

$joinNonce : Nonce$
$svrAddr, devAddr : Addr$

A connection between a device and server (similar to a TCP connecton in Section 5.1) has a unique id *cid* and network identified by endpoints *orig* and *dest*, a connection *mode* that indicates whether connection is requested or accepted, and session keys.

Connection

$cid : Id$
$orig, dest : Addr$
$mode : req \mid acc \mid \ldots$
$sess : Sess$

An end-device has unique *did*, keeps a join server id *jsid* and its last received nonce *joinNonce*, unique root keys r assigned by a manufacturer, and maintains a connection c of which it is the *orig*.

Device

$$
\begin{array}{l}
did, jsid : Id \\
joinNonce : Nonce \\
r : Root \\
c : Connection \\
\hline
r.appKey \neq \bot \\
r.nwkKey \neq \bot \\
c.orig = did \\
c.dest = jsid
\end{array}
$$

We omit details of the gateway and model instead a server which consists of:

- *id*, a unique id, and *addr*, a unique network address;
- *supported*, a database of device ids and their root keys;
- *netAddrs*, a database of device ids and their network address;
- *db*, a database of device ids and their nonces used during handshake; and
- *cnx*, a set of connections with other devices with which it is involved.

Server

$$
\begin{array}{l}
id : Id \\
addr : Addr \\
supported : Id \rightarrow Root \\
netAddrs : Id \rightarrow Addr \\
db : Id \rightarrow Nonce \\
cnx : \mathbb{P}\, Connection \\
\hline
supported \neq \varnothing
\end{array}
$$

7.4 The handshake

Before the start of OTAA, the end-device has no prior knowledge of network addresses and session keys but knows the preshared secret keys, namely the network

key *nwkKey* and application key *appKey*. The server s also knows these keys (recall *Authenticated* in Section 3.2.1). The network key ensures mutual authentication and message integrity while the application key ensures secrecy of session keys. Fig. 7.1 shows the OTAA procedure with interaction occurs between a device A and a server B.

---

1. $A \rightarrow B$: $aid, did, dnon, mic_1$
   B : **if** $mic_1 = H_{nwkKey}(aid.did.dnon) \wedge dnon \notin db$
   then $db := db \cup \{dnon\}$; compute session keys
   else *Error*
2. $B \rightarrow A$: $\{anon +\!\!+ nid +\!\!+ daddr +\!\!+ dnon +\!\!+ extra +\!\!+ mic_2\}_{appKey}$
   A : **if** $mic_2 = H_{nwkKey}(anon +\!\!+ nid +\!\!+ daddr +\!\!+ dnon +\!\!+ extra)$
   then compute session keys
   else *Error*

Figure 7.1: Over-the-air activation (OTAA) handshake between end device A and server B. The notation $+\!\!+$ between strings denotes their concatenation.

In Step 1, A sends to B a join request consisting of its id *did*, the application server *aid* with which it wants to communicate (we abstract this to be the same as the network server id), and a nonce value *dnon*.

When B receives the join request, it verifies that the message has not been tampered with by computing a message integrity code from the contents of the message and comparing the result with mic_1 that arrived with the join request. If verification of mic_1 succeeds and the nonce *dnon* is fresh, B updates A's nonce *dnon* and computes new session keys; otherwise it invokes an error procedure. B now computes new session keys as follows:

$$nwkSKey := H_{appKey}(anon +\!\!+ nid +\!\!+ daddr +\!\!+ dnon), \qquad \text{Definition (7.1)}$$

$$appSKey := H_{appKey}(anon +\!\!+ nid +\!\!+ daddr +\!\!+ dnon). \qquad \text{Definition (7.2)}$$

According to Lemma 2, successful verification means that B knows that A knows the network session key *nwkKey*, $K_B K_A$ *nwkKey*.

In Step 2 B sends to A a join accept encrypted with *appKey* (recall encryption from Section 3.1.4). The join accept consists of its an application nonce *anon*, its network identifier *nid*, an end-device address *daddr*, an acknowledgement of A's nonce *dnon*, *extra* information related to the channel, and a message integrity code mic_2.

A now receives the join accept. Assuming A succeeds in decrypting the message, by Acknowledgement Lemma 1, A now knows that B knows *appKey* used to encrypt the message, $K_A K_B$ *appKey*. If verification succeeds, according to Lemma 2, A now knows that B knows *nwkKey*, $K_A K_B$ *nwkKey*. A then computes the same session keys as in (7.1) and (7.2). If verification of mic_2 fails, an error procedure is invoked. A can also increase its state of knowledge according to Acknowledgement Lemma 1, whenever B acknowledges *dnon* then A knows that B knows *dnon*, i.e., $K_A K_B$ *dnon*.

Lemma 7. *The OTAA handshake achieves mutual authentication*

$$Authenticated(A, B, nwkKey),$$

if communicating parties are honest and do not deviate from OTAA.

Proof According to Lemma 2, the communications in Figure 7.1 establish

$$\begin{aligned} &K_B K_A\, nwkKey && \text{(Step 1)} \\ &K_A K_B\, nwkKey && \text{(Step 2)} \quad , \end{aligned}$$

which establish the first two conjuncts of *Authenticated* (3.4).

Suppose an intruder C eavesdrops communications between A and B. C learns all the information in the join request, and knows the encrypted join accept message, i.e.,

$$K_C\,(aid, did, dnon, mic_1), \tag{7.3}$$

$$K_C\,(\{anon +\!\!+ nid +\!\!+ daddr +\!\!+ dnon +\!\!+ extra +\!\!+ mic_2\}_{appKey}) \quad . \tag{7.4}$$

Because A and B do not reveal any information about $nwkKey$ and $appKey$, an eavesdropper C does not know these values, $\neg\, K_C\,(nwkKey, appKey)$. It is therefore infeasible for C to verify mic_1 in Equation (7.3) without knowledge of the value of $nwkKey$. It is also infeasible for C to decrypt the message in Equation (7.4) without knowledge of $appKey$, and thereafter to verify mic_2 without knowledge of $nwkKey$. □

7.4.1 Operation *JoinReq*

The establishment of a connection between a device D and server S starts with D invoking a join-request $JoinReq$ operation. We split $JoinReq$ into operations request Req at D and receive request $ReqRec$ at S.

Req prepares the join-request message with the following inputs: the server sid, join-request type and a mic. It sends these details to the join server.

$$
\begin{array}{|l}
\hline
Req \\
\hline
\Xi Device \\
sid? : Id \\
req? : Request \\
mic? : \mathrm{seq}\,\mathbb{D} \\
m! : Message \\
\hline
m!.hdr.orig = did \\
m!.hdr.dest = sid? \\
m!.pyl = req? \\
m!.mic = mic? \\
\hline
\end{array}
$$

That operation is total.

Meanwhile, the message $m?$ is received by S with operation $ReqRec$. The server updates its database db only if the device id did and nonce $devNonce$ are new and are supported by the server. It also creates a connection for the device. The server creates a connection with a fresh cid, an $orig = id$ and $dest = did$, derives new session keys, and sets connection $mode$ to indicate this is a connection request. The server then updates its list of connections with the new connection.

ReqRec

$$
\begin{array}{l}
\Delta Server(db, cnx) \\
m? : Message \\
\hline
(m?.pyl.did, m?.pyl.devNonce) \notin db \\
m?.pyl.did \in \operatorname{dom} supported \\
db' = db \cup \{m?.pyl.did \mapsto m?.pyl.devNonce\} \\
\exists\, c : Connnection \cdot \left(\begin{array}{l}
c.cid :\in Id \\
c.orig = sid \\
c.dest = netAddr(m?.pyl.did) \\
c.sess = H(supported(m?.pyl.did).nwkKey, req?) \\
c.mode = req \\
cnx' = cnx \cup \{c\}
\end{array} \right)
\end{array}
$$

The precondition for *ReqRec* is that the device nonce is not already in the database and is supported by the network. The function H computes sessions keys (see Definition (6)).

pre *ReqRec*

$$
\begin{array}{l}
s : Server(db, supported) \\
m? : Message \\
\hline
\exists\, db : s.db,\ sp : s.supported \cdot \left(\begin{array}{l}
m?.pyl.devNonce) \notin db \\
m?.pyl.did \in \operatorname{dom} sp
\end{array} \right)
\end{array}
$$

An internal operation *Verify* checks the message *mic*.

Verify

$$
\begin{array}{l}
Server \\
mic? : \mathbb{D} \\
nwkKey : \mathbb{K} \\
pyl : \operatorname{seq} \mathbb{D} \\
\hline
mic? = H(nwkKey, pyl)
\end{array}
$$

Once verification succeeds, an internal operation *CreateNetAddr* assigns the device a new network address. If verification fails, an error procedure is invoked.

CreateNetAddr

$$\Delta Server(netAddr)$$
$$did? : Id$$

$$did? \notin \operatorname{dom} netAddr$$
$$\exists\, addr : Addr \cdot netAddr' = netAddr \cup \{did? \mapsto addr\}$$

The precondition for *CreateNetAddr* is that the device does not already have a network address.

7.4.2 Operation *JoinAcc*

We assume the server was successful in verifying the *mic* and wishes to continue with the connection. The server responds to the join-request operation with *JoinAcc*, which we split into *AckSend* at S and *AckRec* at the device D.

In *AckSend* the server acknowledges the join request and prepares a join-accept message. The server creates a message with the join server nonce *joinNonce*, the new device address *devAddr* and a *mic* to be verified by the device.

AckSend

$$\Delta Server(cnx)$$
$$joinNonce? : Nonce$$
$$devAddr? : Addr$$
$$mic? : \operatorname{seq} \mathbb{D}$$
$$m! : Message$$

$$\exists\, c : cnx \cdot \begin{pmatrix} c.dest = devAddr? \\ c.mode = req \\ c'.mode = acc \end{pmatrix}$$
$$m!.hdr.orig = sid$$
$$m!.pyl.joinNonce = joinNonce?$$
$$m!.ply.svrAddr = svrAddr$$
$$m!.ply.devAddr = devAddr?$$
$$m!.mic = mic?$$

The precondition for *AckSend* is that there is a connection with destination given as input *devAddr*? with mode equal *req*. The server sends to the device the message

$m!$.

pre *AckSend*

$$
\begin{array}{l}
s : Server(cnx) \\
joinNonce? : Nonce \\
devAddr? : Addr \\
mic? : \operatorname{seq} \mathbb{D}
\end{array}
$$

$$
\exists\, c : s.cnx;\ m! : Message \cdot \left(\begin{array}{l} c.dest = devAddr? \\ c.mode = req \end{array} \right) \land \left(\begin{array}{l} m!.hdr.orig = sid \\ m!.pyl.joinNonce = joinNonce? \\ m!.ply.svrAddr = s.svrAddr \\ m!.ply.devAddr = devAddr? \\ m!.mic = mic? \end{array} \right)
$$

In operation *AckRec* device D receives the join accept message and updates its connection information.

AckRec

$$
\begin{array}{l}
\Delta Device(joinNonce, c) \\
m? : Message
\end{array}
$$

$$
\begin{array}{l}
m?.hdr.dest = did \\
m?.pyl.joinNonce > joinNonce \\
joinNonce' = m?.pyl.joinNonce \\
c'.orig = m?.pyl.devAddr \\
c'.dest = m?.ply.svrAddr \\
c'.mode = acc \\
c'.sess = H(r.nwkKey, m?)
\end{array}
$$

The precondition for *AckRec* is that the incoming message is meant for the device d with incoming *joinNonce* strictly greater than the previous *joinNonce*.

pre *AckRec*

$$d : Device(joinNonce, c)$$
$$m? : Message$$

$$\exists\, conn : Connection \;\cdot \begin{pmatrix} m?.hdr.dest = d.did \\ m?.pyl.joinNonce > d.joinNonce \\ conn = d.c \end{pmatrix}$$

7.5 Key-management vulnerability

An attacker may learn a device's root keys by cryptanalysis or physical capture of the device. It can then impersonate the device and continue to compute session keys. This violates the security requirement of mutual authentication, that only authenticated users know certain shared values and no one else. Furthermore, once computed session keys remain static throughout the lifetime of the device and are reset only during special times, like when a device rejoins the network (see LoRaWAN documentation [1]). This violates the requirement of secrecy. A violation of the protocol's security properties constitutes a vulnerability. Session keys need to be updated frequently in order to preserve these security properties.

7.5.1 Mitigating the vulnerability

In this section we compare a lightweight key-exchange scheme in the literature[76, 77] with one that we design. The former uses elliptic-curve cryptography, to generate small session keys, and a compression algorithm, to create small message packets. We propose a new key-management scheme that uses the approach discussed in Section 4.3 of Chapter 4. Finally we compare the two schemes in terms of communication overhead, and memory requirements.

7.5.1.1 Elliptic-Curve Diffie-Hellman (ECDH) approach

The lightweight mitigation scheme proposed in the literature is called the **E**lliptic-**C**urve **D**iffie-**H**ellman **O**ver **C**oncise Binary Object Representation Object (ED-

HOC) scheme [76, 77]. EDHOC combines elliptic curve cryptography (ECC) and the Diffie-Hellman key exchange to update session keys *nwkSKey*, *appSKey*. ECC has the advantage of generating smaller encryption keys compared with larger Rivest-Shamir-Adleman (RSA) keys with the same security strength [49, 61]. EDHOC has three mandatory messages added to the end of the OTAA handshake. The extra messages increase communication overhead. However the scheme uses a compression technique to decrease messages sizes to 1/6-th the size of a normal LoRaWAN message. Coupled with the smaller ECC keys, this technique conserves memory.

The scheme uses an elliptic curve E over a finite field $\mathbb{F}_p$ with order p (a large prime), and generator g to generate a shared secret. The parameters g and p are common knowledge between a device A and and network server B. Points over the elliptic curve form a group with respect to scalar multiplication [49]. A selects a random private key $a : \mathbb{F}_p$ and computes ephemeral public key ag; similarly for B with b and bg. A then sends ag to B and B sends bg to A. A calculates shared secret $s = (bg)a = (ag)b$, which is what B computes as well by association. The scheme then uses the shared secret s to update session keys. Security of the scheme relies on the infeasibility of computing abg unless an attacker knows either a or b (see Definition (5)).

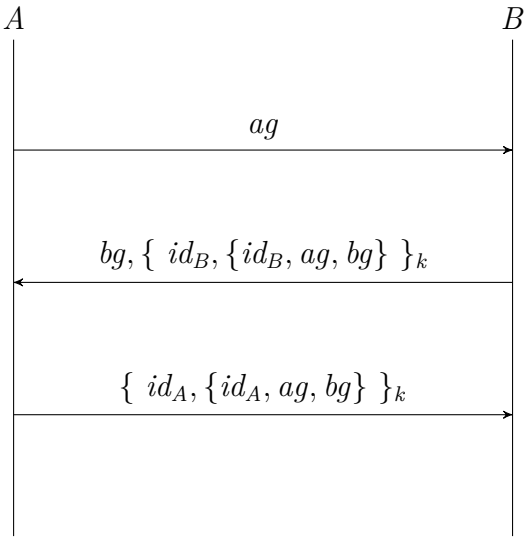

Figure 7.2: The EDHOC scheme. The values $a, b : \mathbb{F}_p$ are private keys belonging to A and B respectively; the values ag and bg are the ECDH ephemeral public keys (recall encryption notation from Section 3.1.4).

7.5.1.2 Commitment approach

A network server periodically sends ping messages to a device to check its status. We propose a scheme *KeyUpdate* that leverages these ping messages. When the server sends a ping, it will also include a prompt to update session keys. This update may be done every hour, every week, etc. depending on power requirements.

Before the ping is sent, the device A and server B have derived initial session keys after the OTAA handshake. They must share hash function h that will be iterated n times to produce new ephemeral network and application session keys. During the ping operation, B sends to A the request for the i-th session key. A receives it and computes new application and network session keys. It creates hashed targets y_a, y_n of the new session keys and sends them to B, which it receives and stores. B has a database that stores hashed targets for each user.

During *Update*, A sends new session keys x_a, x_n to B, which it receives and checks

$$
\begin{array}{rrl}
\mathit{Post\text{-}OTAA} := & A, B: & \mathit{appSKey}, \mathit{nwkSKey}, h, n :\in \mathbb{N} \mid n \geq 2 \\
\mathit{Init} := & B: & i := n - 1 \\
& B \to A: & i \\
& A: & i := n - 1 \fatsemi \\
& & x_a = h^i(\mathit{appSKey}) \fatsemi \\
& & x_n = h^i(\mathit{nwkSKey}) \fatsemi \\
& & y_a := h(x_a) \fatsemi \\
& & y_n := h(x_n) \\
& A \to B: & y_a, y_n \fatsemi \\
& & \mathit{Update} \\
\mathit{Update} := & A: & \textbf{if } i \geq 1 \textbf{ then } A \to B: \quad x_a, x_n \fatsemi \\
& & \qquad\qquad\qquad\qquad\quad i := i - 1 \\
& & \qquad\qquad\quad \textbf{else} \quad \mathit{Init} \\
& B: & \textbf{if } h(x_a) = y_a \wedge h(x_n) = y_n \textbf{ then } B \to A: \mathit{Accept} \fatsemi \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\quad y_a := x_a \fatsemi \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\quad y_n := x_n \fatsemi \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Update} \\
& & \qquad\qquad\qquad\qquad\qquad\quad \textbf{else} \quad B \to A: \mathit{Error}
\end{array}
$$

Figure 7.3: The *KeyUpdate* algorithm, in which B sends a ping message containing a request for the i-th session key to be computed.

that they match its stored values. If they do, B will update its database with the new session keys.

Theorem 11. *OTAA combined with KeyUpate is secure.*

A proof of this theorem follows similar reasoning as the proof for Theorem 7 (see Chapter 4). □

7.5.2 Efficiency

EDHOC requires three additional but mandatory messages to establish session keys with the ECDH approach. Our approach relies on existing LoRaWAN ping messages to trigger the algorithm with i and update the hashed targets y_a, y_n. But

it introduces two additional messages, the hashed targets x_a, x_n sent by A to B and the *Accept* or *Error* message sent by B. *KeyUpdate* has less communication overhead.

In terms of memory requirements, EDHOC and *KeyUpdate* both update session keys of the same size, and therefore use the same amount of memory for storage.

7.6 Conclusion

Key management is vital in ensuring that messages are encrypted end-to-end. It involves updating session keys regularly and hiding them from attackers.

This chapter has analysed the Long Range Wide Area Network (LoRaWAN) protocol and highlighted the functional properties of LoRaWAN, namely the Over-the-air Activation (OTAA) handshake procedure and its join request and join accept operations. We have found that the protocol does achieve mutual authentication but does not have an explicit key-management strategy. Without it, there is no guarantee of forward and future secrecy of session keys. If an attacker compromises the session keys, it can decrypt past and future messages. This constitutes a vulnerability.

We have used our methodology to highlight successfully this vulnerability in the setting of the Internet of Things, and reason about its mitigation. Our mitigation is new and uses the scheme introduced in Chapter 4, Section 4.3. This solution is lightweight in terms of communication overhead and memory requirements.

Chapter 8

Conclusion

8.1 Summary

The current research has developed an approach to analysing security protocols, which combines epistemic logic and the Z specification language. Epistemic logic specifies precisely the security requirements of a protocol in terms of who knows what here extended to feasibility. Z describes state and state transitions. The approach makes certain hardness assumptions about the cryptographic primitives underlying a protocol, and proves that the protocol as result achieves its security goals in the context of the Internet of Things. As a result we have been able to reason about privacy and security as functional properties.

We have used epistemic logic to express functional definitions for the following security properties: mutual authentication; forward secrecy; and future secrecy. The Z schema notation has been used to specify the protocols as abstract data types with state and operations on state. The calculation of preconditions associated with an abstract data type's operations ensure the schemas are consistent. Where possible we have augmented the schemas with epistemic expressions to express what is achieved by the state operations. We have then reasoned algebraically that protocols achieve their security properties given certain assumptions about the cryptographic primitives underlying them, and assumptions about the capabilities

of adversaries that attack the protocols.

To demonstrate workability of our approach, we have applied it to benchmark examples: Diffie-Hellman Key-Exchange Protocol; and Needham-Schroeder-Lowe Protocol. We have then used it on Lamport's One-Time Password Authentication Scheme, for use later in the thesis. In each case we have been able to reason algebraically to show these protocols meet their security goals. In case of standard communications protocols, the methodology has proved it is able to give accounts (specify, reveal flaws and express and reason about their mitigation) of man-in-the-middle and split handshake attacks.

Of particular focus in this thesis was reasoning about security vulnerabilities in lightweight distributed systems like the Internet of Things. Lightweight distributed systems consist of devices with limited compute and memory resources, and so must use these resources efficiently. In order to reason about vulnerabilities we have considered an adversary or attacker that eavesdrops and remembers all communications between agents but its computations are bounded by polynomial-time. Security properties are made possible by cryptographic primitives underlying the protocols. A feature of this thesis is that it focusses on specifying what these cryptographic primitives achieve, not how they work. A case in point, is the assumptions made about the hash function primitive that is used to establish certain hardness properties like the infeasibility of invertability, independent of any particular hash function.

As a result we have further demonstrated our approach on the Signal Protocol, and the Long Range Wide Area Network (LoRaWAN) protocol each capable of operating on devices with limited resources. We have reasoned about security of these protocols under man-in-the-middle attack. In particular we have been able to reason about the key-management vulnerability in LoRaWAN and suggested a mitigation based on Lamport's One-Time Password Authentication Scheme. Correctness of our results follows directly from reasoning about execution of the protocol and reasoning about man-in-the-middle attacks that try to violate mutual authentication

and secrecy properties. In the case of Lamport's One-Time Password Authentication Scheme, the Signal and LoRaWAN protocols, we have provided theoretical evaluations that indicate the feasibility of these schemes for lightweight distributed systems.

8.2 Conclusions

Protocol implementation follows a straightforward approach: define a protocol, write code and test it in the real world. Hackers find ways to exploit weaknesses in protocols by subverting security requirements. So how do we ensure error-free or strong security guarantees in implementations? Whilst confirming the importance of testing this is were our approach comes in. Overall we have been able successfully to state security properties as functional properties and specify the behaviour of security protocols as abstract data types consisting of state and operations. Our proofs have shown that epistemic expressions extended to be true with high probability can be successfully combined with state transitions to analyse the security of protocols designed for lightweight distributed systems. Analysing protocols as abstract data types circumvents the problem of reasoning about different implementations of the same protocol. However, it is challenging to capture enough state such that schema observables and predicates do not become overly complicated.

Little work has been done applying epistemic logic and Z specifications to the Internet of Things protocols. We believe our approach contributes to the analysis of similar security protocols and compliments that of model checking of a single implementation. The epistemic definitions indicate that it is possible to specify security goals as functional properties. However our analysis was applied in a specific context, that of the Internet of Things, and so proof of security is relative to protocols in that context. Therefore it has not been possible to prove unconditionally that security properties hold in other contexts mainly because techniques for generalisation of assumptions do not yet exist. However we believe our epis-

temic definitions are general enough and contribute towards building universally acceptable definitions for security properties.

The methodology proposed in this thesis has been shown to be appropriate for the kinds of case study considered. This is due to its calibration on benchmarks which have provided us with confidence to apply it to a new mitigation for LoRaWAN.

8.3 Future work

To explore further implications, future studies could extend our approach to include more definitions of security properties. By modelling more security properties, it could help make security properties more standard and thus easily included in the analysis and design of security protocols. It would also be interesting to explore the extent to which the epistemic approach could be made quantitative. Future work could also be to examine the practical feasibility of the suggested key-management solution for the Long Range Wide Area Network protocol.

Bibliography

[1] *LoRaWAN Backend Interfaces 1.1 Specification.* `https://lora-alliance.org/resource_hub/lorawan-back-end-interfaces-v1-0/`, 2017. [Online; access 4 May 2019].

[2] M. Abadi and M. R. Tuttle. A logic of authentication. In *ACM Transactions on Computer Systems*. Citeseer, 1990.

[3] M. Adalier. Efficient and Secure Elliptic Curve Cryptography Implementation of Curve P-256. In *Workshop on Elliptic Curve Cryptography Standards*, volume 66, 2015.

[4] L. Alliance. LoRaWAN 1.0.3 Specification. LoRa Alliance Press Release, 2018. [Online; accessed: 7 October 2019].

[5] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. Recommendation for key management part 1: General (revision 3). *NIST Special Publication*, 800(57):1–147, 2012.

[6] E. Bäumker, A. M. Garcia, and P. Woias. Minimizing power consumption of LoRa® and LoRaWAN for low-power wireless sensor nodes. In *Journal of Physics: Conference Series*, volume 1407, page 012092. IOP Publishing, 2019.

[7] T. Beardsley and J. Qian. The TCP Split Handshake: Practical Effects on Modern Network Equipment. *Network Protocols & Algorithms*, 2(1):197–217, 2010.

[8] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Annual International Cryptology Conference*, pages 232–249. Springer-Verlag, 1993.

[9] D. J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer-Verlag, 2006.

[10] M. Blum. Coin flipping by telephone a protocol for solving impossible problems. *ACM SIGACT News*, 15(1):23–27, 1983.

[11] G. Brassard, C. Crépeau, D. Mayers, and L. Salvail. A brief review on the impossibility of quantum bit commitment. *arXiv preprint quant-ph/9712023*, 1997.

[12] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. In *Symposium on Operating Systems Principles*, 1989.

[13] R. Chadha, S. Delaune, and S. Kremer. Epistemic Logic for the Applied Pi Calculus. *Formal Techniques for Distributed Systems*, page 182, 2009.

[14] K. M. Chandy and J. Misra. How processes learn. *Distributed Computing*, 1(1):40–52, 1986.

[15] D. Chaum, J.-H. Evertse, and J. Van De Graaf. An Improved Protocol for Demonstrating Possession of Discrete Logarithms and Some Generalizations. In *Proceedings of the 6th Annual International Conference on Theory and Application of Cryptographic Techniques*, pages 127–141. Springer-Verlag, 1988.

[16] J. A. Clark and J. L. Jacob. A survey of authentication protocol literature: Version 1.0. https://pure.york.ac.uk/portal/en/publications/, 1997. [Online access; 26 May 2021].

[17] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila. A Formal Security Analysis of the Signal Messaging Protocol. In *European Symposium on Security and Privacy (EuroS&P)*, pages 451–466. IEEE, 2017.

[18] K. Cohn-Gordon, C. Cremers, and L. Garratt. On Post-Compromise Security. In *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*, pages 164–178. IEEE, 2016.

[19] C. J. F. Cremers. *Scyther: Semantics and verification of security protocols*. Eindhoven University of Technology Eindhoven, Netherlands, 2006.

[20] F. Dechesne and Y. Wang. To know or not to know: epistemic approaches to security protocol verification. *Synthese*, 177(1):51–76, 2010.

[21] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[22] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[23] R. Duke and G. Rose. *Formal Object-Oriented Specification Using Object-Z*. Macmillan, 2000.

[24] M. Eldefrawy, I. Butun, N. Pereira, and M. Gidlund. Formal security analysis of LoRaWAN. *Computer Networks*, 148:328–339, 2019.

[25] J. H. Ellis. The history of non-secret encryption. *Cryptologia*, 23(3):267–273, 1999.

[26] R. Fagin and J. Y. Halpern. Reasoning about knowledge and probability. *Journal of the ACM*, 41(2):340–367, 1994.

[27] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. An operational semantics for knowledge bases. In *AAAI*, pages 1142–1147, 1994.

[28] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. Common Knowledge Revisited. *Annals of Pure and Applied Logic*, 96(1-3):89–105, 1999.

[29] R. Fagin, Y. Moses, J. Y. Halpern, and M. Y. Vardi. *Reasoning about knowledge*. MIT press, 2003.

[30] U. Feige, A. Fiat, and A. Shamir. Zero Knowledge Proofs of Identity. *Journal of cryptology*, 1(2):77–94, 1988.

[31] T. Frosch, C. Mainka, C. Bader, F. Bergsma, J. Schwenk, and T. Holz. How secure is TextSecure? In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pages 457–472. IEEE, 2016.

[32] J. Fruhlinger. The Mirai botnet explained: How teen scammers and CCTV cameras almost brought down the internet. https://www.csoonline.com/article/3258748/, 2018. [Online access; 26 May 2021].

[33] J. Glasgow, G. MacEwen, and P. Panangaden. A Logic for Reasoning About Security. *ACM Transactions on Computer Systems*, 10(3):226–264, 1992.

[34] S. Goldwasser, S. Micali, and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM Journal on computing*, 18(1):186–208, 1989.

[35] L. C. Guillou and J.-J. Quisquater. A “paradoxical" indentity-based signature scheme resulting from zero-knowledge. In *Conference on the Theory and Application of Cryptography*, pages 216–231. Springer, 1988.

[36] J. Halpern, Y. Moses, and M. Tuttle. A knowledge-based analysis of zero knowledge. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 132–147, 1988.

[37] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990.

[38] J. Y. Halpern, R. Pass, and V. Raman. An epistemic characterization of zero knowledge. In *Proceedings of the 12th Conference on Theoretical Aspects of Rationality and Knowledge*, pages 156–165, 2009.

[39] J. Y. Halpern and M. R. Tuttle. Knowledge, probability, and adversaries. *Journal of the ACM (JACM)*, 40(4):917–960, 1993.

[40] J. Hamill. Samsung spy telly scandal erupts after firm admits its television will RECORD your "personal and sensitive" conversations. `https://www.mirror.co.uk/news/technology-science/technology/samsung-spy-telly-scandal-erupts-5131842`, 2015.

[41] J. Hintikka. *Knowledge and Belief*. Ithaca: Cornell University Press, 1962.

[42] C. A. R. Hoare. Communicating Sequential Processes. In *The Origin of Concurrent Programming*, pages 413–443. Springer-Verlag, 1978.

[43] M. Hung. *Leading the IoT Garnter Insights on How to Lead a Connected World*, 2017.

[44] T. Hunt. No, VTech cannot simply absolve itself of security responsibility. `https://www.troyhunt.com/no-vtech-cannot-simply-absolve-itself/`, 2016.

[45] M. N. Kamkuemah. On Discrete Dynamical Systems. In *Workshop for African Women in Discrete Mathematics and its Applications January 2018 AIMS South Africa*, January 2018. Monograph in preparation.

[46] M. N. Kamkuemah. Epistemic Analysis of a Key-Management Vulnerability in LoRaWAN. In *2021 18th International Conference on Privacy, Security and Trust (PST)*, pages 1–7, 2021.

[47] M. N. Kamkuemah. Reasoning about Authentication and Secrecy in the Signal Protocol. In *2021 International Conference on Electrical, Computer and Energy Technologies (ICECET)*, pages 1–6, 2021.

[48] M. N. Kamkuemah. Zero-Knowledge Authentication. In *Federated Africa and Middle East Conference on Software Engineering Earth 2022 (FAMESCE)*. ACM, 2022. Submitted.

[49] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. CRC press, 2008.

[50] J. Katz, A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[51] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas. DDoS in the IoT: Mirai and other botnets. *Computer*, 50(7):80–84, 2017.

[52] H. Krawczyk. Cryptographic Extraction and Key Derivation: The HKDF Scheme. In *Annual Cryptology Conference*, pages 631–648. Springer-Verlag, 2010.

[53] S. A. Kripke. A completeness theorem in modal logic. *The Journal of Symbolic Logic*, 24(1):1–14, 1959.

[54] L. Lamport. Password Authentication with Insecure Communication. *Communications of the ACM*, 24(11):770–772, 1981.

[55] L. Lamport. *Specifying Systems: the TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[56] G. Lowe. An Attack on the Needham- Schroeder Public- Key Authentication Protocol. *Information Processing Letters*, 56(3), 1995.

[57] G. Lowe. A Hierarchy of Authentication Specifications. In *Proceedings 10th Computer Security Foundations Workshop*, pages 31–43. IEEE, 1997.

[58] G. Lowe. A family of attacks upon authentication protocols. Technical report, 1997.

[59] M. Malinspike. There is no WhatsApp 'backdoor'. https://signal.org/blog/there-is-no-whatsapp-backdoor/, 2017. [Online; access 28 October 2019].

[60] M. Marlinspike and T. Perrin. The X3DH key agreement protocol. *Open Whisper Systems*, 2016.

[61] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC press, 2018.

[62] S. Messenger. Open Whisper Systems: Technical Information, Specifications. `https://signal.org/blog/advanced-ratcheting/`, 2018. [Online; access 10 June 2021].

[63] C. Morgan. The shadow knows: Refinement of ignorance in sequential programs. In *International Conference on Mathematics of Program Construction*, pages 359–378. Springer, 2006.

[64] C. Morgan. The shadow knows: Refinement and security in sequential programs. *Science of Computer Programming*, 74(8):629–653, 2009.

[65] K. Munro. Making children's toys swear. `https://www.pentestpartners.com/security-blog/making-childrens-toys-swear/`, 2015.

[66] C. Nachreiner. What is the TCP split-handshake attack and does it affect me? `https://www.secplicity.org/2011/04/15/what-is-the-tcp-split-handshake-attack-and-does-it-affect-me/`, April 2011.

[67] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[68] J. M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32(143):918–924, 1978.

[69] C. Pomerance. A tale of two sieves. *Notices American Mathematical Society*, 43:1473–1485, 1996.

[70] J. Postel. RFC 793: Transmission Control Protocol. `https://datatracker.ietf.org/doc/html/rfc793`, 1981. [Online access; 26 May 2021].

[71] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[72] A. W. Roscoe. Modelling and verifying key-exchange protocols using csp and fdr. In *Proceedings The Eighth IEEE Computer Security Foundations Workshop*, pages 98–107. IEEE, 1995.

[73] R. Sanchez-Iborra, J. Sánchez-Gómez, S. Pérez, P. Fernández, J. Santa, J. Hernández-Ramos, and A. Skarmeta. Enhancing lorawan security through a lightweight and authenticated key management approach. *Sensors*, 18(6):1833, 2018.

[74] M. Santos and A. Faure. Affordance is power: Contradictions between communicational and technical dimensions of whatsapp's end-to-end encryption. *Social Media+ Society*, 4(3):2056305118795876, 2018.

[75] C.-P. Schnorr. Efficient Signature Generation by Smart Cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[76] G. Selander, J. Mattsson, and F. Palombini. Ephemeral Diffie-Hellman Over COSE (EDHOC). `https://datatracker.ietf.org/doc/html/draft-selander-ace-cose-ecdhe-13`, 2019. Work in Progress.

[77] G. Selander, J. Mattsson, and F. Palombini. Ephemeral Diffie-Hellman over COSE (EDHOC). *IETF, ID draft-selander-lake-edhoc-00*, 2019.

[78] P. Sethi and S. R. Sarangi. Internet of things: architectures, protocols, and applications. *Journal of Electrical and Computer Engineering*, 2017, 2017.

[79] G. Smith. Principles of secure information flow analysis. In *Malware Detection*, pages 291–307. Springer-Verlag, 2007.

[80] M. A. S. Smith. *Formal verification of TCP and T/TCP*. PhD thesis, Massachusetts Institute of Technology, 1997.

[81] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., 1989.

[82] J. M. Spivey and J. Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

[83] H. Suo, J. Wan, C. Zou, and J. Liu. Security in the internet of things: a review. In *2012 international conference on computer science and electronics engineering*, volume 3, pages 648–651. IEEE, 2012.

[84] P. F. Syverson and P. C. Van Oorschot. On unifying some cryptographic protocol logics. In *Proceedings of 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 14–28. IEEE, 1994.

[85] A. S. Tanenbaum and D. J. Wetherall. *Computer Networks, fifth edition*. Prentice Hall, 2011.

[86] B. Toxen. *Real World Linux Security: Intrusion Prevention, Detection, and Recovery*. Prentice Hall Professional, 2003.

[87] H. M. Zeidler. End-to-end encryption system and method of operation, 1986. US Patent 4,578,530.