**Project IST-1999-11583**

**Malicious- and Accidental-Fault Tolerance for Internet Applications**


# Final report on verification and assessment

André Adelsbach and Sadie Creese (Editors)

**MAFTIA deliverable D22**

Public document

January 27, 2003

**Editors**

André Adelsbach ..... *Universität des Saarlandes (D)*
Sadie Creese ..... *QinetiQ (UK)*

**Contributors**

André Adelsbach ..... *Universität des Saarlandes (D)*
Sadie Creese ..... *QinetiQ (UK)*
Richard Harrison ..... *QinetiQ (UK)*
Birgit Pfitzmann ..... *IBM Zurich Research Lab (CH)*
Ahmad-Reza Sadeghi ..... *Universität des Saarlandes (D)*
William Simmonds ..... *QinetiQ (UK)*
Michael Steiner ..... *Universität des Saarlandes (D)*
Christian Stüble ..... *Universität des Saarlandes (D)*
Michael Waidner ..... *IBM Zurich Research Lab (CH)*

**Abstract**

MAFTIA workpackage 6 is concerned with the rigorous definition of core MAFTIA concepts, and the verification and assessment of the work on dependable middle-ware. In the former MAFTIA deliverables D4 [1] and D8 [2], we presented general rigorous models for the security of synchronous and asynchronous reactive systems. These models comprised various types of faults (attacks) and topology as considered in MAFTIA, and we provided two example models of secure message transmission. Furthermore, we proved composition theorems for both models which facilitates modular proof. In deliverables D4 [1] and D7 [3] we used the process algebra CSP and the FDR model checker to verify MAFTIA services running in synchronous and asynchronous systems respectively.

In this deliverable we model group key establishment protocols and present a rigorous security proof. We chose these protocols because secure group key establishment is an important prerequisite for MAFTIA's dependable middle-ware, services and applications. The example demonstrates the trusted host model's applicability for proving more complex cryptographic protocols by making use of the composition theorem. Furthermore, it illustrates the modelling of adaptive corruptions in the secure reactive system models.

We then investigate how to 'bridge the gap' between the rigorous secure reactive systems theory of Pfitzmann and Waidner [4] and the world of automated model checking using CSP and FDR. We model the group key agreement protocol, and discuss how properties asserted of the trusted host can be transcribed in CSP. In Chapter 4 we describe our work on verifiying the Trusted Timely Computing Base (TTCB), a core MAFTIA concept. We examine two services: The Local Authentication Service and Trusted Block Agreement Service.

Finally, we continue our analysis of the security impact of the abstractions made in the secure reactive system models, when implementing secure reactive systems in the real world (see [2]). Concluding our analysis, we discuss measures to implement these abstractions in the real world and outline a secure real world execution environment for secure reactive systems.

# Contents


**1 Introduction** **1**
- 1.1 Objectives of MAFTIA Work-package 6 . . . 1
- 1.2 Deliverable Structure . . . 1
- 1.3 CSP, $\text{CSP}_M$ and FDR . . . 3

**2 Formal Model for Multi-party Group Key Agreement** **6**
- 2.1 Introduction . . . 6
- 2.2 Basic Definitions and Notation . . . 7
  - 2.2.1 System Model and Simulatability . . . 8
  - 2.2.2 Standard Cryptographic Systems . . . 10
  - 2.2.3 Notation . . . 11
- 2.3 Ideal System for Group Key Establishment . . . 12
- 2.4 Real System for Group Key Establishment . . . 21
  - 2.4.1 Initial Key Agreement . . . 21
- 2.5 Security of Real System . . . 27
  - 2.5.1 Generalized Diffie-Hellman Machine . . . 28
  - 2.5.2 Real System Rewritten with Interactive Diffie-Hellman Machine . . . 35
  - 2.5.3 Replacing $\mathsf{GDH}^{(0)}_{n,mxkey}$ by $\mathsf{GDH}^{(1)}_{n,mxkey}$ . . . 45
  - 2.5.4 Security with Respect to the Ideal System . . . 45
- 2.6 Conclusion . . . 48

**3 CSP Model of GKE** **50**
- 3.1 Background . . . 50
- 3.2 Overview of Model . . . 52
- 3.3 Datatypes . . . 54
- 3.4 Trusted Host state variables . . . 55
- 3.5 Functions . . . 57
- 3.6 Channels . . . 57
- 3.7 $KEYR$ . . . 60
- 3.8 $VARIABLES$ . . . 62
- 3.9 $\text{CSP}_M$ model of the Trusted Host state-transition machine . . . 66
- 3.10 $\text{CSP}_M$ model of the Ideal System . . . 72
- 3.11 $\text{CSP}_M$ transcriptions of two group key establishment properties 73

3.11.1 Key Secrecy . . . 73
3.11.2 PFS . . . 74
3.12 Running the Scripts . . . 75
3.13 Report . . . 76

**4 CSP Modelling Of Selected TTCB Security Services 77**
4.1 Background . . . 77
4.1.1 The Local Authentication Service . . . 79
4.1.2 The Trusted Block Agreement Service . . . 80
4.2 Datatypes . . . 81
4.3 The Local Authentication Service . . . 85
4.3.1 The Payload Network . . . 85
4.3.2 The Entity and Local TTCB . . . 88
4.3.3 CSP transcription of the informal properties . . . 94
4.3.4 Running the Scripts . . . 96
4.3.5 Initial Results . . . 97
4.3.6 Results . . . 99
4.3.7 Improvements . . . 101
4.4 The Trusted Block Agreement Service . . . 102
4.4.1 Design Decisions . . . 102
4.4.2 The local TTCB Node . . . 105
4.4.3 CSP transcription of the informal properties . . . 116
4.4.4 Running the Scripts . . . 118
4.4.5 Results . . . 119
4.4.6 Improvements . . . 121

**5 Faithfully Implementing Reactive Systems 123**
5.1 Introduction . . . 123
5.2 Possible Approaches . . . 124
5.3 Impacts on Real-World Security . . . 125
5.3.1 Trust Model and Corruption Model . . . 125
5.3.2 Enforcement of Interfaces . . . 126
5.3.3 Communication . . . 130
5.3.4 Implementing Links . . . 131
5.3.5 Implementing Ports . . . 133
5.3.6 Concurrency . . . 133
5.3.7 Additional Information Flows . . . 133

5.3.8 Type Enforcement . . . . . 136
5.4 A Secure Execution Environment for Reactive Systems . . . . 136
5.4.1 The IT-Environment . . . . . 137
5.4.2 The Operating System . . . . . 137
5.4.3 Middle Layer . . . . . 139
5.4.4 TCPA and Palladium . . . . . 141
5.5 Conclusion . . . . . 142

**6 Final Comment** **143**

**7 Appendix** **145**
Bibliography . . . . . 147

# 1 Introduction

## *1.1 Objectives of MAFTIA Work-package 6*

The MAFTIA project systematically investigates the tolerance paradigm for building dependable distributed systems. For this, it combines techniques and notions from fault tolerance and various areas of security, such as intrusion detection and cryptographic protocols.

The goal of the Work-package 6 in the MAFTIA project is to enable the verification and assessment of the dependability achieved by protocols and mechanisms developed in the other work-packages. To meet this goal, we have developed rigorously defined models (see D4, D7 and D8 [1, 3, 2]) that cover the basic concepts identified in Work-package 1 (see D2 and D21 [5, 6]). In deliverables D4 and D7 we used the process algebra CSP and the FDR model checker to verify MAFTIA services running in synchronous and asynchronous systems respectively.

Furthermore, we have begun to investigate how to link cryptographic and formal-methods approaches to proving complex systems secure, in a manner which makes complete proof as easy as is possible: proofs that allow abstraction and the use of formal methods, but retain a sound cryptographic semantics. Our rigorous system models allow us to split reactive systems into two layers: The lower layer is a cryptographic system whose security can be rigorously proven using standard cryptographic arguments. To the upper layer it provides an abstract (and typically deterministic) service that hides all cryptographic details. Relative to this abstract service one can verify the upper layer using existing formal methods. Since our model allows secure composition one can conclude that the overall system is secure if the formally verified upper layer is put on top of a cryptographically verified lower layer. An example for this approach has been shown in [7].

## *1.2 Deliverable Structure*

We start this deliverable by modelling group key establishment protocols and presenting a rigorous security proof for these protocols in Chapter 2. We chose these protocols because secure group key establishment is an important prerequisite for MAFTIA's dependable middle-ware, services

and applications. It demonstrates the model's applicability for proving the correctness of more complex cryptographic protocols by making use of the composition theorem: the proof separates cryptographic, i.e., computational and probabilistic parts, like the Diffie-Hellman decision problem, from the deterministic state keeping parts of the protocols. Furthermore, it illustrates the modelling of adaptive corruptions in the secure reactive system models.

In Chapter 3 we present another, yet different, example of linking cryptography and formal methods based on the secure reactive system models. The Trusted Host specification of the secure group key establishment service is a quite complex state machine and, therefore, potentially prone to mis-specification. We modelled the group key establishment Trusted Host in CSP and considered how we may use the model to prove, using FDR, that certain informally stated security properties hold of the Trusted Host. If successful, that would increase the level of confidence in the Trusted Host being a correct specification for group key establishment.

In Chapter 4 we present our modelling of two security services offered by a core MAFTIA concept: the Trusted Timely Computing Base (TTCB). We present an argument as to why the Local Authentication service of correct for any number of local TTCBs and entities. We verify the Trusted Block Agreement Service for a fixed number of entities.

Finally, in Chapter 5 we continue our discussion of the abstractions from the real world made in the presented models as presented in deliverable D8 [2]. D8 mainly focused on assessing the abstractions' impact on real world security of implementations in standard runtime environments. It was the focus of our ongoing research to investigate a corresponding implementation architecture, including tools and guidelines, which support an implementor in constructing a secure real-world system. In this deliverable we present our findings of measures to implement these abstractions in the real world and outline a secure real world execution environment for secure reactive systems, which should help implementors in faithfully implementing secure reactive systems in the real world. We close this deliverable in Chapter 6 with our closing comment.

## 1.3 CSP, $CSP_M$ and FDR

For part of our assessment of MAFTIA concepts we have chosen to utilise the process algebra CSP[8, 9] and the FDR[10] model checker[1]. Recent advances in the theory of CSP have enabled the model checking of systems of unbounded size. For certain models we can gain the benefits of both model checking (support for fault removal) and theorem proving (proofs independent of parameter sizes, such as network size and topology). This makes the CSP/FDR combination an ideal candidate for the verification and assessment of the novel synchrony models and complex protocols being developed in the MAFTIA project.

The CSP user community has been modelling security protocols using standard synchrony models almost since the arrival of FDR, about nine years ago. There have been many success stories where ambiguities and faults have been found. The recent book by Lowe, Goldsmith, Roscoe, Ryan and Schneider [11] gives a detailed description of the CSP modelling approach, and is also a good source of examples.

CSP is a process algebra which is useful for describing systems that interact by communication. A system is modelled as a *process* (itself possibly constructed from a collection of processes) which interacts with its environment by means of atomic *events*. Communication is synchronous; an event takes place precisely when both the process and the environment agree on its occurrence. The syntax of CSP provides a variety of operators for modelling processes, and the associated algebra provides rewrite laws. Primitive operators include process prefix, sequential composition, deterministic and non-deterministic choice, parallelism, interleaving, hiding, recursion, deadlock and successful termination:

$$\begin{aligned} P \ ::= \ & a \rightarrow P \mid P \,;P \mid P \,\Box\, P \mid P \sqcap P \mid P \parallel P \mid P \,|||\, P \mid \\ & P \setminus b \mid \mu X \bullet F(X) \mid STOP \mid SKIP \end{aligned}$$

The collection of mathematical models and associated semantics that make up CSP facilitate the capture of a wide range of process behaviours. The traces model captures the observable traces of events which a CSP process might exhibit. The failures model captures not only the traces of a process, but also those events it can refuse to perform after a particular trace.

[1] FDR takes as its input syntax a machine readable version of CSP.

The failures divergences model also captures information about events which a process may diverge on (perform infinitely often) from a particular state.

The theory of refinement in CSP allows correctness conditions to be encoded as refinement checks between processes. If process $P$ refines process $Q$, then all of the possible behaviours of $P$ must also be possible behaviours of $Q$ (although $Q$ may also possess many other behaviours). Therefore, $P$ is a correct, and more deterministic, implementation of $Q$. This notion of refinement holds for all three of the semantic models, where the possible behaviours of processes are interpreted in terms of the semantic model under consideration.

Refinement is transitive:

$$R \sqsubseteq S \wedge S \sqsubseteq T \Rightarrow R \sqsubseteq T$$

If process $R$ is refined by process $S$ (written $R \sqsubseteq S$), and $S$ is refined by $T$ then $R$ is refined by $T$. All CSP operators are monotonic with respect to refinement:

$$R \sqsubseteq T \Rightarrow C[R] \sqsubseteq C[T]$$

where $C[]$ is a context built from CSP operators and constants. This facilitates compositional development of systems. Imagine we want to prove:

$$Spec \sqsubseteq C[System]$$

we can break this into two parts: Find a process which does refine the desired specification:

$$Spec \sqsubseteq C[P]$$

and prove that:

$$P \sqsubseteq System$$

By monotonicity:

$$C[P] \sqsubseteq C[System]$$

and by transitivity:

$$Spec \sqsubseteq C[P] \wedge C[P] \sqsubseteq C[System] \Rightarrow Spec \sqsubseteq C[System]$$

The FDR tool takes a machine-readable dialect of $\mathrm{CSP}_M$ as its input syntax, and can be used to check refinements as well as determinism, deadlock

freedom and livelock freedom of processes. $\text{CSP}_M$ is the combination of a rich data language, based on functional programming, and the CSP process algebra. The various $\text{CSP}_M$ operators which are used in this thesis are given in Appendix 7. See [10, 9] for detailed descriptions of $\text{CSP}_M$.

# 2 Formal Model for Multi-party Group Key Agreement

## *2.1 Introduction*

Most group-oriented applications, like MAFTIA's middleware, service replication mechanisms and auhtorisation service (see deliverables [12, 13, 14]), depend on the ability of all group members to securely multicast messages to all others. Thus *group key establishment* is an important building block for such applications [15]. Essentially there are two ways to establish a shared group key, *group key distribution,* where a trusted party chooses and distributes the key, and *group key agreement.*

Group key agreement generalizes the standard 2-party key agreement, as introduced in [16, 17]: Each group member knows the membership of the group (a so-called peer group) and has a public key known to everybody. At the end of a successful protocol run each member obtains a randomly chosen group session key. No key information must be leaked to anybody outside the group. The adversary might adaptively corrupt parties, obtaining single keys or all information that has not been explicitly erased before the corruption. Groups are often dynamic, i.e., members might join or leave the group. *Auxiliary* group key agreement protocols enable the members of an updated group to agree on a new group key more efficiently than performing a new group key agreement from scratch [18].

Two-party key agreement has been intensively studied and several precise definitions and provably secure protocols have been developed [19, 20, 21, 22]. (However, none of them is in a model for which a composition theorem has been proved, i.e., that the protocol remains secure in arbitrary environments; hence our definition is also beneficial for the two-party case.) For group key agreement, several protocols have been proposed, e.g., [17, 23, 24, 18, 25, 26, 27, 28], but no precise definition existed so far and thus no group key agreement protocol has been rigorously proven. Past work on formalizing group key protocols is either limited in scope [26] (key distribution only, no key agreement) or still work-in-progress [29, 30]; the latter also suffers from the fundamental problems [31] of abstracting cryptography with a term algebra [32]. Parallel yet independent to our work [33] proposed very recently a formal definition based on on the formalization tradition of [19] and proof

of protocols very similar to ours.

In the following, we give a precise definition of group key agreement in a general reactive simulatability-based model (see, e.g., [2] or [34]): Essentially, we specify an ideal group key agreement system where a single, non-corruptible party $\mathsf{TH}$, called *trusted host*, serves all parties. Whenever a group wants to agree on a new key, $\mathsf{TH}$ chooses a random key and distributes it to all group members, provided they are all non-corrupted. Depending on when a member becomes corrupted, $\mathsf{TH}$ gives the random key to the adversary $\mathsf{A}$ or lets $\mathsf{A}$ even choose the keys for the non-corrupted parties. We assume an asynchronous network completely controlled by the adversary. A real system is considered secure if whatever happens to the honest users in this real system with some adversary $\mathsf{A}$ could happen to the same honest users in the ideal system, with some other adversary $\mathsf{A}'$.

Our definition covers all informal security notions like *key authentication* and *forward secrecy*. It also covers *auxiliary protocols*. Furthermore, these properties persist under arbitrary composition with higher-level protocols [35].

Based on these definitions we analyze the security of concrete protocols. We start with a protocol from [36] for a network with authenticated channels. We also derive a second protocol to handle adaptive corruptions. We then prove them secure against static and adaptive adversaries, respectively. In the process, we investigate the concrete security of the $n$-party Diffie-Hellman decision problem and the generation of a random string from the obtained group element, and we prove the security of an interactive version of this problem.

## *2.2 Basic Definitions and Notation*

Our definitions and proofs are based on the notion of standard cryptographic systems with adaptive corruptions as defined in MAFTIA deliverable D8 [2], Section 2.4. We briefly recapitulate this model, omitting all details not needed in this deliverable.

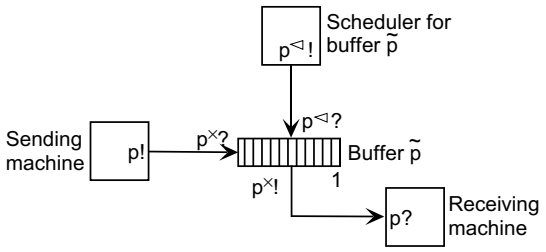


Figure 2.1: Ports and buffers

### 2.2.1 System Model and Simulatability

The systems are parametrized with a **security parameter**, $k \in \mathbb{N}$, and depend on the number of participants, $n \in \mathbb{N}$. Let $\mathcal{M} := \{1, \ldots, n\}$.

The main component of a system is a set of **protocol machines**, $\{\mathsf{M}_1, \ldots, \mathsf{M}_n, \mathsf{Gen}\}$ for real systems and $\{\mathsf{TH}\}$ for ideal systems. Intuitively, $\mathsf{M}_u$ serves user $u$. Machine $\mathsf{Gen}$ is incorruptible; it is used for reliably generating and distributing initial parameters used by all machines (in our case a cyclic group and generator for the Diffie-Hellman setting and a corresponding universal hash function to map GDH keys to bit strings).

The machines are *probabilistic state-transition machines* (where the state-transition functions are realized by arbitrary probabilistic *Turing machines*.) Each machine can communicate with other machines via ports. **Output (input) ports** are written as $\mathsf{p}!$ ($\mathsf{p}?$), and $\mathsf{Ports}(\mathsf{M})$ denotes the set of all ports of a machine $\mathsf{M}$. Messages are transported from $\mathsf{p}!$ to $\mathsf{p}?$ over a connection represented by a **buffer** machine $\widetilde{\mathsf{p}}$. A buffer $\widetilde{\mathsf{p}}$ stores all messages received from $\mathsf{p}!$ at $\mathsf{p}^{\times}?$ and waits for inputs on its **clock port** $\mathsf{p}^{\triangleleft}?$. Each input $i \in \mathbb{N}$ triggers $\widetilde{\mathsf{p}}$ to put the $i$-th stored message on $\mathsf{p}^{\times}!$ (or no message if it contains less than $i$ messages) to be forwarded to $\mathsf{p}?$. Ports and buffers are illustrated in Figure 2.1.

A **structure** is a pair $(M, S)$, where $M$ is a set of machines and $S$, the **specified ports**, is a subset of the free ports[1] of the union of $M$ and all the buffer machines needed for connections used or clocked by machines in $M$. $S$ models the service interfaces offered or required by $M$. The remaining free ports will be available to the adversary and model unavoidable or tolerable

[1] **Free ports** of a set of machines are all input (output) ports $\mathsf{p}?$ ($\mathsf{p}!$) where the corresponding output (input) port $\mathsf{p}!$ ($\mathsf{p}?$) is not associated to any machine in the set.

adversary control and information flow to and from the adversary. This is often required — even in an ideal system — to achieve realistic models without further unwanted restriction, e.g., for a practical key establishment protocol there is normally no harm when the adversary learns who runs the protocol with whom. Nonetheless, without modeling this information flow in a trusted host, a faithful implementation of that trusted host would have to be based on a (costly) anonymous network.

A structure describes (known) components and their interaction with the (unknown) environment. However, to obtain a whole runnable system we have to specify the environment, too. Therefore, the structure $(M, S)$ is complemented to a **configuration** by adding an arbitrary **user machine** $\mathsf{H}$, which abstracts higher-layer protocols and ultimately the end user, and an **adversary machine** $\mathsf{A}$. $\mathsf{H}$ connects to ports in $S$ and $\mathsf{A}$ to the rest, and they may interact. We will describe the specified ports not directly but by their *complements*, $S^c$, i.e., by listing the ports that $\mathsf{H}$ should have. Finally, a **system** *Sys* is a set of structures.

The machines in a configuration are scheduled sequentially: In principle only buffers have clock input ports, like $\mathsf{p}^{\triangleleft}?$ for buffer $\widetilde{\mathsf{p}}$. The currently active machine $\mathsf{M}_s$ can schedule any buffer $\widetilde{\mathsf{p}}$ for which it owns $\mathsf{p}^{\triangleleft}!$, and if $\widetilde{\mathsf{p}}$ can actually deliver a message, this schedules the receiving machine $\mathsf{M}_r$. If $\mathsf{M}_s$ tries to schedule multiple buffers at a time then only one is taken, and if no buffer is scheduled (or the scheduled one cannot deliver a message) then a designated **master scheduler** is scheduled; usually, the adversary $\mathsf{A}$ plays that role. A configuration is a runnable system, i.e., one gets a probability space of runs and views of individual machines in these runs.

**Simulatability** essentially means that whatever can happen to certain users in the real system can also happen to the same users in the ideal system: for each configuration $(M, S, \mathsf{H}, \mathsf{A})$ there is a configuration $(\{\mathsf{TH}\}, S, \mathsf{H}, \mathsf{A}')$ such that the views of $\mathsf{H}$ in the two configurations are indistinguishable [37]. Simulatability is abbreviated by "$\geq_{\mathsf{sec}}$." As by definition only good things can happen in the ideal system, simulatability guarantees that no bad things can happen in the real world.

### 2.2.2 Standard Cryptographic Systems

In a **standard cryptographic system** with static adversaries, *Sys* is a set of structures $(M_{\mathcal{H}}, S_{\mathcal{H}})$, one for each set $\mathcal{H} \subset \mathcal{M}$ of non-corrupted users. The structures $(M_{\mathcal{H}}, S_{\mathcal{H}})$ are derived from an **intended structure** $(M^*, S^*)$, where $M^* = \{\mathsf{M}_1^*, \ldots, \mathsf{M}_n^*\}$, $S^{*c} = \{\mathsf{in}_u!, \mathsf{in}_u{}^{\triangleleft}!, \mathsf{out}_u? \mid u \in \mathcal{M}\}$ and $\{\mathsf{in}_u?, \mathsf{out}_u!, \mathsf{out}_u{}^{\triangleleft}!\} \subseteq \mathsf{Ports}(\mathsf{M}_u^*)$. Each $S_{\mathcal{H}}$ is the subset of $S^*$ where $u$ only ranges over $\mathcal{H}$.[2] The derivation depends on a **channel model**: Each connection (i.e., buffer) of $(M^*, S^*)$ is labeled as "**secure**" (private and authentic), "**authenticated**" (only authentic), or "**insecure**" (neither authentic or private.) In the derivation all output ports of authenticated connections are duplicated; thus $\mathsf{A}$ connects to them and can read all messages. All insecure connections are routed through $\mathsf{A}$, i.e., the ports are renamed so that both become free and thus connected to $\mathsf{A}$. The reliability of a connection is implicitly determined by the definition of specified ports: If the clock output port corresponding to a buffer is a specified port, we have a **reliable**, otherwise an **unreliable** channel.

For adaptive adversaries,[3] the derivation makes some additional modifications: The specified ports are extended[4] by ports $\{\mathsf{corrupt}_u!, \mathsf{corrupt}_u{}^{\triangleleft}! \mid u \in \mathcal{M}\}$ used for corruption requests.[5] Furthermore, each $\mathsf{M}_u$ gets three additional ports $\mathsf{corIn}_u?$, $\mathsf{corOut}_u!$ and $\mathsf{corOut}_u{}^{\triangleleft}!$ for communication with $\mathsf{A}$ after corruption: If $\mathsf{M}_u$ receives $(\mathsf{do})$ on $\mathsf{corrupt}_u?$ in state $\sigma$ it encodes $\sigma$ in some standard way and outputs $(\mathsf{state}, \sigma)$ at $\mathsf{corOut}_u!$ (i.e., reveals everything it knows to $\mathsf{A}$). From then on it is taken over by $\mathsf{A}$ and operates in **transparent mode**: Whenever $\mathsf{M}_u$ receives an input $m$ on a port $\mathsf{p}? \neq \mathsf{corIn}_u?$, it outputs $(\mathsf{p}, m)$ at $\mathsf{corOut}_u!$. Whenever it receives an input $(\mathsf{p}, m)$ on $\mathsf{corIn}_u?$ for which $\mathsf{p}!$ is a port of $\mathsf{M}_u$, it outputs $m$ at that port. Over time any subset of $\{\mathsf{M}_1, \ldots, \mathsf{M}_n\}$ can become corrupted.[6]

[2] Consequently, for each set $\mathcal{H}$ one trusted host $\mathsf{TH}_{\mathcal{H}}$ is defined.

[3] For a more concise presentation and without loss of generality, we slightly deviate from [2] and [34]: We use a separate structure for each set $\mathcal{H} \subset \mathcal{M}$ also for the adaptive case even though a single structure for $\mathcal{M}$ would have sufficed.

[4] If those names are already occupied they can be renamed arbitrarily.

[5] Those must be made via specified ports as the service will change at the corresponding ports $\mathsf{in}_u?$ and $\mathsf{out}_u!$ also in the ideal system.

[6] In terms of [34]: our adversary structure is $\mathcal{ACC} = 2^{\{\mathsf{M}_1, \ldots, \mathsf{M}_n\}}$.

### 2.2.3 Notation

Variables are written in italics (like *var*), constants and algorithm identifiers in straight font (like const and algo), and sets of users in calligraphic font (like $\mathcal{M}$). For a set $set \subseteq \mathbb{N}$ and $i \leq |set|$, let $set[i]$ denote the $i$-th element with respect to the standard order $<$ on $\mathbb{N}$ and $\mathsf{idx}(set, elem)$ the index of *elem* in *set* if present and $-1$ otherwise, i.e., $\mathsf{idx}(set, set[i]) = i$ for $i \in \{1, \ldots, |set|\}$. By $I_n := \{0,1\}^n$ we denote the set of all $n$-bit strings.

Machines are specified by defining their state variables and transitions. The variable *state* of $\mathsf{M}_i$ is written as $\mathsf{M}_i.state$, or, if clear from the context such as in a transition rule, as *state* only. To simplify notation, we allow arrays that range over an infinite index set, like $(a_i)_{i\in\mathbb{N}}$, but always initialize them everywhere with the same value (e.g., undef for "undefined"). Thus, they can be efficiently represented by polynomial-time machines.

Transitions are described using a simple language similar to the one proposed in [38]. Most of the notation should be clear without further explanations. Each transition starts with "**transition** $\mathsf{p}?(m)$" where $\mathsf{p}?$ is an input port and $m$ an abstract message, i.e., a message template with free variables. Optional parameters in $m$ are denoted by $[\ldots]$ and their presence can be queried using the predicate $\mathsf{present}(\cdot)$. An "**enabled if:** *cond*" (where *cond* is an arbitrary boolean expression on machine-internal state variables) specifies the condition under which the transition is enabled. If the (optional) **enabled if:** is absent, the transition is always enabled. When a message msg arrives at a port $\mathsf{p}?$ and all transitions on this port are disabled, the message is silently (and at no computational cost[7] for the corresponding machine) discarded. Otherwise, we first increment the **message counter** $\mathsf{p}?.cntr$ associated with the given input port $\mathsf{p}?$. This counter keeps track of the number of activations on a port (and indirectly the computational cost of a machine)

[7]Recall that the condition *cond* of **enabled if:** depends only on machine-internal state variables. This allows the computation to be done on state-changes and requires no computation on message arrival. For example, if the condition also would depend on the message, a real-time evaluation (and, hence, computation costs) would be required on message arrival. This would make such a construct unsuitable for the context discussed here, i.e., specifying how ports can be disabled such that messages on these ports do not incur any computational costs. Nevertheless, as such broader conditions are useful in other cases, there is a second similar construct "**ignore if:** *cond*" where *cond* may also depend on variables of the message.

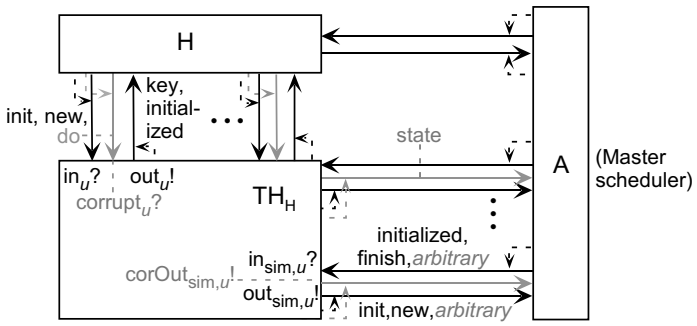


Figure 2.2: Trusted host and its message types. Parts related to adaptive adversaries are in gray. Dashed lines indicate who schedules a connection.

and is initialized to zero. If the message msg matches the template $m$ of any enabled transition on this port, the corresponding transition is executed. Without loss of generality, we further require from the specification that at any given time at most one enabled transition matches any given message. The final states of a machine are implicitly defined as the situations when no transition is enabled anymore.

## *2.3 Ideal System for Group Key Establishment*

The following scheme specifies the trusted host for an ideal system for group key establishment.

**Scheme 2.1 (Ideal System for Group Key Establishment $Sys^{\mathsf{gke,ideal}}_{n,tb,ct}$)**
Let $n, tb \in \mathbb{N}$ and $ct \in \{\mathsf{static}, \mathsf{adaptive}\}$ be given, and let $\mathcal{M} := \{1, \ldots, n\}$. Here, $n$ denotes the number of intended participants, $tb$ a bound on the number of activations per port — this is required to make $\mathsf{TH}$ polynomial — and $ct$ the type of corruptions to be tolerated. An ideal system for secure group key establishment is then defined as

$$Sys^{\mathsf{gke,ideal}}_{n,tb,ct} = \{(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}) \mid \mathcal{H} \subseteq \mathcal{M}\}.$$

Here $\mathcal{H}$ denotes the set of a priori uncorrupted participants. Let $\mathcal{A} := \mathcal{M} \setminus \mathcal{H}$.

An overview of $\mathsf{TH}_{\mathcal{H}}$ is given in Figure 2.2. The ports of $\mathsf{TH}_{\mathcal{H}}$ are $\{\mathsf{in}_u?, \mathsf{out}_u!, \mathsf{out}_u{}^{\triangleleft}!, \mathsf{corrupt}_u?, \mathsf{in}_{\mathsf{sim},u}?, \mathsf{out}_{\mathsf{sim},u}!, \mathsf{out}_{\mathsf{sim},u}{}^{\triangleleft}!, \mathsf{corOut}_{\mathsf{sim},u}!, \mathsf{corOut}_{\mathsf{sim},u}{}^{\triangleleft}! \mid u \in \mathcal{H}\}$. The specified ports are as described in Section 2.2.2

for standard cryptographic systems, i.e., $S^{*\,c}_{\mathcal{H}} = \{\mathsf{in}_u!, \mathsf{in}_u{}^{\triangleleft}!, \mathsf{out}_u? \mid u \in \mathcal{H}\}$ and $S^{c}_{\mathcal{H}} = S^{*\,c}_{\mathcal{H}} \cup \{\mathsf{corrupt}_u!, \mathsf{corrupt}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}$.

The message formats exchanged over the ports are shown in Table 2.1. Common parameters are as follows: $u \in \mathcal{M}$ identifies a user, $grp \subseteq \mathcal{M}$ is the set of group members, $sid \in \mathcal{SID}$ is a session identifier (relative to a group $grp$), and $key \in \{0,1\}^k$ is an exchanged session key. The domain of session identifiers, $\mathcal{SID}$, can be arbitrary as long as the representations of elements can be polynomially bounded in $k$ (otherwise resulting machines might not be polynomial anymore.)

The state of $\mathsf{TH}_{\mathcal{H}}$ is given by the variables shown in Table 2.2. The state-transition function is defined by the following rules; the message types are also summarized in Figure 2.2.

**Initialization.** Assume an honest $u \in \mathcal{M}$, i.e., one with $u \in \mathcal{H}$ and $\mathsf{TH}_{\mathcal{H}}.state_{u,u} \neq \mathsf{corrupted}$. $\mathsf{H}$ triggers initialization of $u$ by entering $\mathsf{init}$ at $\mathsf{in}_u?$. In a real system, $\mathsf{M}_u$ will now set system parameters, generate long-term keys, etc., and possibly send public keys to other machines. In the ideal system $\mathsf{TH}_{\mathcal{H}}$ just records that $u$ has triggered initialization by the state $\mathsf{wait}$. Any subsequent input $\mathsf{init}$ is ignored. The adversary immediately learns that $u$ is initializing. (In most real systems initialization requires interaction with other machines, which is visible to the adversary.)

**transition** $\mathsf{in}_u?$ $(\mathsf{init})$
  **enabled if:** $(state_{u,u} = \mathsf{undef}) \wedge (\mathsf{in}_u?.cntr < tb)$;
  $state_{u,u} \leftarrow \mathsf{wait}$;
  **output:** $\mathsf{out}_{\mathsf{sim},u}!$ $(\mathsf{init})$, $\mathsf{out}_{\mathsf{sim},u}{}^{\triangleleft}!$ $(1)$;
**end transition**

By entering $(\mathsf{initialized}, v)$ at $\mathsf{in}_{\mathsf{sim},u}?$ the adversary triggers that an honest user $u$ learns that user $v$, potentially $u$ itself, has been initialized. Note that this can happen even before $u$ has been initialized itself.

This transition is only enabled when user $u$ is not corrupted and the port's transition bound is not exceeded. The first condition is necessary to disambiguate between this ("honest"-mode) transition and transparent mode after a corruption, i.e., the last two transitions below. The second condition helps making the machine polynomial-time. Both conditions are also part of the enable condition of all other "honest"-mode transitions.

| Port | Type | Parameters | Meaning |
|---|---|---|---|
| *At specified ports $S_{\mathcal{H}}$ to user $u \in \mathcal{H}$* | | | |
| $\mathsf{in}_u?$ | init | | Initialize user $u$. |
| $\mathsf{out}_u!$ | initialized | $v$ | User $v$ initialized from user $u$'s point of view. |
| $\mathsf{in}_u?$ | new | $sid$,$grp$,[$sid$',$grp$'] | Initialize a new session, extending a previous one if optional parameters are present. |
| $\mathsf{out}_u!$ | key | $sid$,$grp$,$key$ | Return newly agreed key. |
| $\mathsf{corrupt}_u?$ | do | | Corrupt user $u$! |
| $\mathsf{out}_u!$ | *arbitrary* | *arbitrary* | Possible outputs after corruptions |
| *At adversary ports* | | | |
| $\mathsf{out}_{\mathsf{sim},u}!$ | init | | User $u$ is initializing. |
| $\mathsf{in}_{\mathsf{sim},u}?$ | initialized | $v \in \mathcal{M}$ | User $u$ should consider user $v$ as initialized. |
| $\mathsf{out}_{\mathsf{sim},u}!$ | new | $sid$,$grp$,[$sid$',$grp$'] | User $u$ has initialized a new session. |
| $\mathsf{in}_{\mathsf{sim},u}?$ | finish | $sid$,$grp$,[$key_{u,sim}$] | Complete session for user $u$. If present and allowed, assign $key_{u,sim}$ to user $u$. |
| $\mathsf{corOut}_{\mathsf{sim},u}!$ | state | *state* | State of corrupted party. |
| $\mathsf{out}_{\mathsf{sim},u}!$ | *arbitrary* | *arbitrary* | Corrupted party $u$ sent a message. |
| $\mathsf{in}_{\mathsf{sim},u}?$ | *arbitrary* | *arbitrary* | Send message to (corrupted) party $u$. |

Table 2.1: The message types and parameters handled by $\mathsf{TH}_{\mathcal{H}}$

| Name | Domain | Meaning | Init. |
|---|---|---|---|
| $(state_{u,v})_{u,v\in\mathcal{M}}$ | $\{\mathsf{undef}, \mathsf{wait}, \mathsf{init}, \mathsf{corrupted}\}$ | Long-term states as seen by user $u$ | undef |
| $(ses_{u,sid,grp})_{u\in\mathcal{M},sid\in\mathcal{SID},grp\subseteq\mathcal{M}}$ | $\{\mathsf{undef}, \mathsf{init}, \mathsf{finished}\}$ | State of sessions as seen by user $u$ | undef |
| $(key_{u,sid,grp})_{u\in\mathcal{M},sid\in\mathcal{SID},grp\subseteq\mathcal{M}}$ | $\{0,1\}^k \cup \{\mathsf{undef}\}$ | Session keys still in negotiation | undef |
| $(prev_{u,sid,grp})_{u\in\mathcal{M},sid\in\mathcal{SID},grp\subseteq\mathcal{M}}$ | $(sid' \in \mathcal{SID}, grp' \subseteq \mathcal{M})$ | Dependency graph of sessions | $(0,\{\})$ |
| $(p?.cntr)_{p\in\{\mathsf{in}_u,\mathsf{corrupt}_u,\mathsf{in}_{\mathsf{sim},u} \mid u\in\mathcal{H}\}}$ | $\mathbb{N}$ | Activation counters | 0 |

Table 2.2: Variables in $\mathsf{TH}_{\mathcal{H}}$

**transition** $\mathsf{in}_{\mathsf{sim},u}?$ (initialized, $v$)
  **enabled if:** $(state_{u,u} \neq \mathsf{corrupted}) \wedge (\mathsf{in}_{\mathsf{sim},u}?.cntr < tb)$;
  **ignore if:** $((state_{v,v} = \mathsf{undef}) \wedge (v \notin \mathcal{A}))\ \vee\ ((u = v) \wedge (state_{u,u} \neq \mathsf{wait}))$;
  $state_{u,v} \leftarrow \mathsf{init}$;
  **output:** $\mathsf{out}_u!$ (initialized, $v$), $\mathsf{out}_u{}^{\triangleleft}!$ (1);
**end transition**

**Group key establishment.** To start a group key establishment for user $u$, $\mathsf{H}$ enters $(\mathsf{new}, sid, grp, [sid', grp'])$ at $\mathsf{in}_u?$. User $u$ has to be a member of the intended group and has to believe that all group members are initialized. Furthermore, the pair $(sid, grp)$ has to be fresh, i.e., never used by $u$ before $(\mathsf{TH}_{\mathcal{H}}.ses_{u,sid,grp} = \mathsf{undef})$; otherwise the command is ignored. The optional parameter $(sid', grp')$ points to a previous group key establishment to which the current one is auxiliary. If $(sid', grp')$ is present, it is required that either $u \notin grp'$ (i.e., this member is added), or the old establishment has terminated and the previous group key was delivered $(\mathsf{TH}_{\mathcal{H}}.ses_{u,sid',grp'} = \mathsf{finished})$. The pair $(sid', grp')$ is recorded in $\mathsf{TH}_{\mathcal{H}}.prev_{u,sid,grp}$. In the real system, $\mathsf{M}_u$ would

now start the protocol. $\mathsf{TH}_{\mathcal{H}}$ just records this fact ($\mathsf{TH}_{\mathcal{H}}.ses_{u,sid,grp} \leftarrow \mathsf{init}$). The adversary immediately learns over port $\mathsf{out}_{\mathsf{sim},u}?$ that $u$ has started an establishment with parameters $sid, grp, [sid', grp']$.

**transition** $\mathsf{in}_u?$ ($\mathsf{new}$, $sid, grp, [sid', grp']$)
  **enabled if:** $(state_{u,u} \neq \mathsf{corrupted}) \wedge (\mathsf{in}_u?.cntr < tb)$;
  **ignore if:** $(u \notin grp) \vee (|grp| < 2) \vee (\exists v \in grp : state_{u,v} \neq \mathsf{init}) \vee$
    $(ses_{u,sid,grp} \neq \mathsf{undef}) \vee$
    $(\mathsf{present}(sid', grp') \wedge (u \in grp') \wedge (ses_{u,sid',grp'} \neq \mathsf{finished}))$;
  $ses_{u,sid,grp} \leftarrow \mathsf{init}$;
  **if** $\mathsf{present}(sid', grp')$ **then**
    $prev_{u,sid,grp} \leftarrow (sid', grp')$;
  **end if**;
  **output:** $\mathsf{out}_{\mathsf{sim},u}!$ ($\mathsf{new}$, $sid, grp, [sid', grp']$), $\mathsf{out}_{\mathsf{sim},u}{}^{\triangleleft}!$ (1);
**end transition**

The adversary decides to finish the protocol for $u$ by entering ($\mathsf{finish}$, $sid$, $grp, [key_{u,sim}]$) at $\mathsf{in}_{\mathsf{sim},u}?$. This input is allowed only once for each honest $u \in grp$. Its effect depends on the presence of dishonest users in $grp$:

- If a group member is dishonest (a priori not in $\mathcal{H}$ or adaptively corrupted) then the adversary can propose[8] a key which $\mathsf{TH}_{\mathcal{H}}$ takes and stores in $\mathsf{TH}_{\mathcal{H}}.key_{u,sid,grp}$. Thus, we do not require anything in this case.

- The same happens if two honest group members do not agree on the previous group. This consistency condition is very weak; e.g., we do not require that the old group was non-corrupted, or that all non-corrupted members obtained the same key. Thus, some protocols for auxiliary key establishment might satisfy only accordingly restricted definitions.

[8]It is essential that passing a key is optional. Otherwise, no protocol providing PFS could be proven secure against adaptive corruptions: Consider a key establishment among two honest users $u$ and $v$ such that $u$ finishes the protocol first and then gets corrupted before $v$ can finish. Such a situation is unavoidable in our asynchronous systems. Since in the real world $u$ and $v$ would have agreed on a common key ($u$ was corrupted only after the session establishment!), the simulator has to model this also in the ideal world. However, this cannot be simulated as we cannot provide $\mathsf{TH}_{\mathcal{H}}$ with the correct key to finish $v$'s session: $u$'s key was generated secretly by $\mathsf{TH}_{\mathcal{H}}$ and not leaked on corruption (it was previously deleted inside $\mathsf{TH}_{\mathcal{H}}$ to make PFS possible.)

- Otherwise, the system will produce a good key, i.e., one chosen randomly from $\{0,1\}^k$. Thus if $u$ is the first group member for which the adversary inputs "finish" (i.e., $\mathsf{TH}_{\mathcal{H}}.ses_{v,sid,grp} \neq \mathsf{finished}$ for all $v \in grp$), then $\mathsf{TH}_{\mathcal{H}}$ selects a good key and stores it for all group members $v$ in $\mathsf{TH}_{\mathcal{H}}.key_{v,sid,grp}$.

The selected key $\mathsf{TH}_{\mathcal{H}}.key_{u,sid,grp}$ is output to $u$, deleted internally ($\mathsf{TH}_{\mathcal{H}}.key_{u,sid,grp} \leftarrow \mathsf{undef}$) (this models forward secrecy), and the key establishment is finished for $u$ ($\mathsf{TH}_{\mathcal{H}}.ses_{u,sid,grp} \leftarrow \mathsf{finished}$).

**transition** $\mathsf{in}_{\mathsf{sim},u}?$ ($\mathsf{finish}$, $sid, grp, [key_{u,sim}]$)
**enabled if:** $(state_{u,u} \neq \mathsf{corrupted}) \wedge (\mathsf{in}_{\mathsf{sim},u}?.cntr < tb)$;
**ignore if:** $(ses_{u,sid,grp} \neq \mathsf{init})$;
**if** $\mathsf{present}(key_{u,sim}) \wedge$
$((\exists v \in grp : state_{v,v} = \mathsf{corrupted} \vee v \in \mathcal{A}) \vee$
$(\exists v_0, v_1 \in grp : (ses_{v_0,sid,grp} \neq \mathsf{undef}) \wedge (ses_{v_1,sid,grp} \neq \mathsf{undef}) \wedge$
$(prev_{v_0,sid,grp} \neq prev_{v_1,sid,grp}))$ **then**
*# Corrupted or inconsistent session so ...*
$key_{u,sid,grp} \leftarrow key_{u,sim}$;*# ... use session key provided by adversary*
**else if** $(\forall\, v \in grp : ses_{v,sid,grp} \neq \mathsf{finished})$ **then**
*# First to finish (ideal) session*
$key \xleftarrow{\mathcal{R}} \{0,1\}^k$; *# Generate new (random) session key ...*
**for all** $v \in grp$ **do**
$key_{v,sid,grp} \leftarrow key$; *# ... and assign it to all parties*
**end for**;
**end if**;
**output:** $\mathsf{out}_u!$ ($\mathsf{key}$, $sid, grp, key_{u,sid,grp}$), $\mathsf{out}_u{}^{\triangleleft}!$ (1);*# Give key to user ...*
$key_{u,sid,grp} \leftarrow \mathsf{undef}$; *# ... and delete it locally to enable forward secrecy*
$ses_{u,sid,grp} \leftarrow \mathsf{finished}$;
**end transition**

**Corruptions.** Corruptions are handled as sketched in Section 2.2.2. A priori, the users in $\mathcal{H}$ are uncorrupted. If $ct = \mathsf{static}$, any inputs on port $\mathsf{corrupt}_u?$ are ignored. If $ct = \mathsf{adaptive}$ then $\mathsf{H}$ can corrupt user $u \in \mathcal{H}$ at any time by entering $\mathsf{do}$ at $\mathsf{corrupt}_u?$. (We do not pose any limitation on the number of users that can be corrupted.) In this case, $\mathsf{TH}_{\mathcal{H}}$ extracts all data corresponding to $u$

with a call to $\mathsf{encode\_state}(u)$ and sends them to $\mathsf{A}$. More precisely, $\mathsf{encode\_state}(u)$ maps to $(\{(u, v, state_{u,v}) \mid v \in \mathcal{M}\}, \{(sid, grp, ses_{u,sid,grp}, key_{u,sid,grp}, prev_{u,sid,grp}) \mid sid \in \mathcal{SID} \wedge grp \subseteq \mathcal{M} \wedge ses_{u,sid,grp} \neq \mathsf{undef}\})$.

The main part are all group keys that are already determined but not yet output to $u$ (and thus not deleted). $\mathsf{TH}_{\mathcal{H}}$ records $u$'s corruption ($\mathsf{TH}_{\mathcal{H}}.state_{u,u} \leftarrow \mathsf{corrupted}$), and from now on operates in transparent mode in respects to ports $\mathsf{in}_u?$ (routed to $\mathsf{out}_{\mathsf{sim},u}!$) and $\mathsf{in}_{\mathsf{sim},u}?$ (routed to $\mathsf{out}_u!$). Note that the transparent mode of the trusted host is slightly different to the transparent mode of standard systems as described in Section 2.2.2. For $\mathsf{TH}_{\mathcal{H}}$, the messages should *not* contain any port indicator: on the one hand, it is always implicitly clear from which input port a message comes or to which output port it has to go, and, on the other hand, explicit port indicators would make the construction of simulators difficult if not impossible.

**transition** $\mathsf{corrupt}_u?$ (do)
  **enabled if:** $(ct = \mathsf{adaptive} \wedge state_{u,u} \neq \mathsf{corrupted})$;
  $state_{u,u} \leftarrow \mathsf{corrupted}$;
  **output:** $\mathsf{corOut}_{\mathsf{sim},u}!$ (state, $\mathsf{encode\_state}(u)$), $\mathsf{corOut}_{\mathsf{sim},u}{}^{\triangleleft}!$ (1);
**end transition**

**transition** $\mathsf{in}_u?$ $(any\_msg)$
  **enabled if:** $(state_{u,u} = \mathsf{corrupted})$; # *Transparent mode*
  **output:** $\mathsf{out}_{\mathsf{sim},u}!$ $(any\_msg)$, $\mathsf{out}_{\mathsf{sim},u}{}^{\triangleleft}!$ (1);
**end transition**

**transition** $\mathsf{in}_{\mathsf{sim},u}?$ $(any\_msg)$
  **enabled if:** $(state_{u,u} = \mathsf{corrupted})$; # *Transparent mode*
  **output:** $\mathsf{out}_u!$ $(any\_msg)$, $\mathsf{out}_u{}^{\triangleleft}!$ (1);
**end transition**

$\diamond$

Let us briefly discuss, why the ideal system defined by Scheme 2.1 matches the notion and properties of a secure group key establishment.This match as well as the preservation of integrity and confidentiality properties by simulation-based proofs allows us to deduce from a proof $Sys^{\mathsf{gke,real}} \geq_{\mathsf{sec}} Sys^{\mathsf{gke,ideal}}_{n,tb,ct}$ (with $Sys^{\mathsf{gke,real}}$ any real-world protocol) that $Sys^{\mathsf{gke,real}}$ inherits all properties from the ideal system and, therefore, is a secure group key es-

tablishment protocol. There are three questions to answer on the model: (1) does it provide an appropriate service, (2) does it capture necessary security properties, and (3) does it support the required group dynamics?

**Service.** It is obvious that the model provides the service "establishment of a common session key." Furthermore, the provided service is as general as possible. To capture all types of key establishment protocols, e.g., (centralized) key transport protocol as well as (distributed) key agreement protocols, the service is independent of particularities of protocols. In particular, it provides a uniformly distributed bit string as key which is the most general abstraction of a key. This is in sharp contrast, e.g., to the model provided by [33] which is highly customized towards Diffie-Hellman-based key agreement protocols.

**Security Properties.** The primary security property to consider is *key secrecy*. For uncorrupted sessions — we cannot expect any secrecy for corrupted sessions — the session key is generated randomly and secretly by $\mathsf{TH}_{\mathcal{H}}$. Furthermore, the adversary will not learn any information on the key other than what is leaked by the users of the key-establishment protocol.[9] This is the strongest secrecy requirement imaginable and also implies the *semantic security* of the session key. The *freshness* of the group key is guaranteed as well since $\mathsf{TH}_{\mathcal{H}}$ generates the session keys randomly and independently from each other.

Except for corruptions, $\mathsf{TH}_{\mathcal{H}}$ returns a session key only to legitimate members of a group. Therefore, the ideal system provides *implicit key authentication*. Additionally, the ideal system ensures that all honest group members successfully establishing an uncorrupted session agree on the same key and know the involved identities. This holds for the following reasons: (1) $\mathsf{TH}_{\mathcal{H}}$ enforces the uniqueness of a session as identified by the pair $(sid, grp)$, (2) this identifier implies a common agreement on the group membership of a session, and (3) $\mathsf{TH}_{\mathcal{H}}$ provides all parties with the same key. As a consequence, the ideal system also offers *mutual group key authentication*. Note that the ideal system does not ensure *explicit group key authentication*. However, this can be easily achieved with following change in **transition** $\mathsf{in}_{\mathsf{sim},u}$? ($\mathsf{finish}$,

[9] This leakage is modeled as flows from $\mathsf{H}$ to $\mathsf{A}$ and is unavoidable when we allow arbitrary modular composition with other protocols.

$sid, grp, [key_{u,sim}]$): replace the condition **ignore if:** $(ses_{u,sid,grp} \neq \mathsf{init})$ by **ignore if:** $(\nexists v \in grp : state_{v,v} = \mathsf{corrupted} \vee v \in \mathcal{A}) \wedge (\exists v \in grp : (ses_{v,sid,grp} = \mathsf{undef}))$, i.e., ensure that for uncorrupted sessions a finish message is handled only when everybody has initialized the session.

The ideal system captures both *PFS* and *KKA*. PFS is addressed by allowing participants to be corrupted: This leaks as part of the state, on the one hand, all their long-term keys but, on the other hand, no session keys from completed sessions. The possibility of KKA is inherent in the model as H can leak arbitrary information to A.

The model does not cover the special properties of (*contributory*) *key agreement* protocols, e.g., the guarantee of key freshness even in sessions with dishonest group members. While these properties are very useful in achieving flexible group key establishment protocols for dynamic peer groups, their security value per se is of only secondary importance and often not required. Therefore, these aspects are not captured in the main model for the sake of a broader model, i.e., one which captures key establishment in general. If desired, however, the model could be extended accordingly, e.g., by adding a restrictions on the freshness of the key passed by the simulator in finish.

**Dynamic groups.** If we omit all the optional arguments $[sid', grp']$ in Scheme 2.1 we obtain the basic notion of group key establishment. In some applications it seems natural to transform one or more existing groups $(sid_1, grp_1), (sid_2, grp_2), \ldots$ into a new group $(sid, grp)$. Forming a group "from scratch" is called *initial key agreement (IKA),* and forming it utilizing existing groups is called *auxiliary key agreement (AKA)* [18]. Any IKA protocol is also an AKA protocol (one that ignores all existing groups), i.e., this distinction is only interesting for performance (number of messages, rounds, etc.).

A group can *grow* by adding a subset $grp_2$ to a group $(sid_1, grp_1)$ via input $(\mathsf{new}, sid, grp_1 \cup grp_2, sid_1, grp_1)$. If $|grp_2| = 1$, this is called *member addition*, otherwise *mass join.*[10] A group can shrink by *excluding* a member $u$ from a group $(sid', grp')$ via input $(\mathsf{new}, sid, grp' \setminus \{u\}, sid', grp')$.

[10]There is also *group join* which transforms two groups $(sid_1, grp_1)$ and $(sid_2, grp_2)$ into $(sid, grp \leftarrow grp_1 \cup grp_2)$. The current ideal system cannot express this directly.

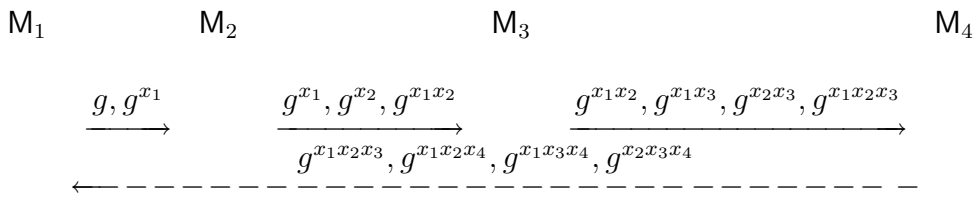


Figure 2.3: Example of IKA.1. (The dotted line denotes a broadcast. The $g$ in the first message could be omitted, but allows a more unified description.)

## *2.4 Real System for Group Key Establishment*

We now consider the security of concrete group key establishment or, more precisely, group key agreement protocols. While some group key agreement protocols from the literature turn out to be secure in a simulatability sense, none does so against adaptive corruptions. We show how to extend them to achieve adaptive security. Both of the following protocols presuppose authenticated connections.

### 2.4.1 Initial Key Agreement

As the basis of the real system, we take the protocol IKA.1 from [18]. (IKA.2 and any other protocol within the framework of the generic protocol from that paper should work in exactly the same way.) An example protocol run is shown in Figure 2.3. In the second line, the so-called downflow phase, the message is broadcasted to every participant so that they can compute the key as $\mathsf{h}(g^{x_1x_2x_3x_4})$.

For the non-adaptive case ($ct = \mathsf{static}$) the protocol is identical to IKA.1 from [18] with the exception of the key computation. Instead of taking the Group Diffie-Hellman key directly, we derive a key using a universal hash-function $\mathsf{h}$ similar to [22]: This is required to get uniformly distributed random bit-strings as keys as mandated by the model, i.e., the ideal system.

For adaptive security ($ct = \mathsf{adaptive}$), we ensure that all secrets have been erased before the first key is output (following [22] for the 2-party case). As long as we use the authenticated channels only, without additional signatures,

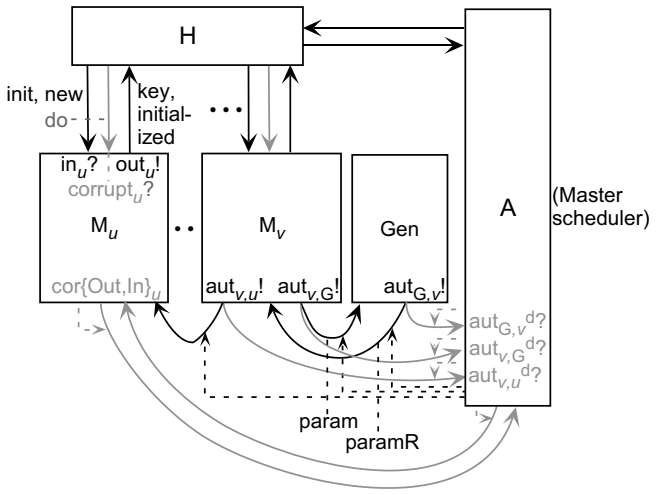


Figure 2.4: Sketch of the real system. Derived parts are shown in gray. Scheduling is shown only for newly introduced ports.

this means a synchronization based on confirmation messages between all pairs of participants.

**Scheme 2.2 (Real System for Group Key Establishment $Sys^{\mathsf{gke,ika1}}_{n,tb,ct}$)**

Let $n \in \mathbb{N}$ be the number of intended participants and $\mathcal{M} := \{1, \ldots, n\}$. Similar to the trusted host, we parameterize the protocol with $tb$, the maximum number of transitions per port, and $ct \in \{\mathsf{adaptive}, \mathsf{static}\}$ depending on whether it has to deal with adaptive adversaries or not. The system $Sys^{\mathsf{gke,ika1}}_{n,tb,ct}$ — see Figure 2.4 for an overview — is defined by the following intended structure $(M^*, S^*)$ and channel model. The actual system is derived as a standard cryptographic system as defined in Section 2.2.2.

The specified ports $S^*$ are the same as in the ideal system, i.e., those connecting user machines $\mathsf{M}$ to $\mathsf{H}$ in Figure 2.4. The intended machines are $M^* = \{\mathsf{M}^*_u \mid u \in \mathcal{M}\} \cup \{\mathsf{Gen}\}$. Their ports are $\mathsf{ports}(\mathsf{M}^*_u) := \{\mathsf{in}_u?, \mathsf{out}_u!, \mathsf{out}_u{}^{\triangleleft}!\} \cup \{\mathsf{aut}_{v,u}?, \mathsf{aut}_{u,v}! \mid v \in \{\mathsf{G}\} \cup \mathcal{M} \setminus \{u\}\}$ and $\mathsf{ports}(\mathsf{Gen}) := \{\mathsf{aut}_{u,\mathsf{G}}?, \mathsf{aut}_{\mathsf{G},u}! \mid u \in \mathcal{M}\}$. All system-internal connections are labeled "authenticated". (Connections to $\mathsf{H}$ are secure.)

The machine $\mathsf{Gen}$ generates and distributes the system parameters. These parameters are generated using the *generation algorithm* $\mathsf{genG}$. On input $1^k$, this algorithm outputs a tuple $(G, g, \mathsf{h})$ where $G$ is a suitable cyclic group of

order $|G|$,[11] $g$ a generator of $G$ and $\mathsf{h}$ a random element of a family $UHF_{G,k}$ of universal hash functions [39] with domain $G$ and range $\{0,1\}^k$. Suitable means that the group operations are efficiently computable, $|G| \geq 2^{3k}$ and the Decisional Diffie-Hellman problem is assumed to be hard.[12]

The machine $\mathsf{Gen}$ is incorruptible, i.e., it always correct. It contains variables $state \in \{\mathsf{undef}, \mathsf{init}\}$ and $(G, g, \mathsf{h})$. Its single state-transition function is:

**transition** $\mathsf{aut}_{u,\mathsf{G}}?$ ($\mathsf{param}$)
  **enabled if:** $(\mathsf{aut}_{u,\mathsf{G}}?.cntr < tb)$;
  **if** $(state = \mathsf{undef})$ **then**
    $(G, g, \mathsf{h}) \leftarrow \mathsf{genG}(1^k)$;
    $state \leftarrow \mathsf{init}$;
  **end if**
  **output:** $\mathsf{aut}_{\mathsf{G},u}!$ ($\mathsf{paramR}$, $G, g, \mathsf{h}$);
**end transition**

A machine $\mathsf{M}_u^*$ implements the group key establishment service for the corresponding user $u$. It contains the variables shown in Table 2.3. Its state-transition function is shown below.

**transition** $\mathsf{in}_u?$ ($\mathsf{init}$) # *Trigger initialization*
  **enabled if:** $(state_u = \mathsf{undef}) \wedge (\mathsf{in}_u?.cntr < tb)$;
  $state_u \leftarrow \mathsf{wait}$;
  **output:** $\mathsf{aut}_{u,\mathsf{G}}!$ ($\mathsf{param}$);
**end transition**

**transition** $\mathsf{aut}_{\mathsf{G},u}?$ ($\mathsf{paramR}$, $G', g', \mathsf{h}'$) # *Get system parameters*
  **enabled if:** $(state_u = \mathsf{wait})$;
  $state_u \leftarrow \mathsf{init}$;
  $(G, g, \mathsf{h}) \leftarrow (G', g', \mathsf{h}')$;
  **output:** $\mathsf{out}_u!$ ($\mathsf{initialized}$, $u$); $\mathsf{out}_u{}^{\triangleleft}!$ (1);
  **for all** $v \in \mathcal{M} \setminus \{u\}$ **do**
    **output:** $\mathsf{aut}_{u,v}!$ ($\mathsf{initialized}$);

[11] The group order $|G|$ and its factorization is assumed to be public. However, for simplicity this is not explicitly coded it into $\mathsf{genG}$'s return.

[12] For the Diffie-Hellman problem to be hard, $|G|$ has to be at least $2^{2k}$ and should not contain any small prime factors [40].

| Name | Domain | Meaning | Init. |
|---|---|---|---|
| $(state_v)_{v\in\mathcal{M}}$ | $\{\mathsf{undef}, \mathsf{wait}, \mathsf{init}, \mathsf{corrupted}\}$ | Long-term states as seen by $\mathsf{M}_u^*$. | undef |
| $(G, g, \mathsf{h})$ | Range of $\mathsf{genG}(1^k)$ | Global parameters. | — |
| $(ses_{sid,grp})_{sid\in\mathcal{SID},grp\subseteq\mathcal{M}}$ | $\{\mathsf{undef}, \mathsf{upflow}, \mathsf{downflow}, \mathsf{confirm}, \mathsf{finished}\}$ | State of a (potential) session. | undef |
| $(\mathcal{C}_{sid,grp})_{sid\in\mathcal{SID},grp\subseteq\mathcal{M}}$ | $\{\mathcal{I} \mid \mathcal{I} \subseteq \mathcal{M}\}$ | Records received session confirmations | $\emptyset$ |
| $(key_{sid,grp})_{sid\in\mathcal{SID},grp\subseteq\mathcal{M}}$ | $\{0,1\}^k \cup \{\mathsf{undef}\}$ | Group key of a session. | undef |
| $(x_{sid,grp})_{sid\in\mathcal{SID},grp\subseteq\mathcal{M}}$ | $\mathbb{Z}_{\lvert G\rvert} \cup \{\mathsf{undef}\}$ | Individual secret key of a session. | undef |
| $(\mathsf{aut}_{v,u}?.cntr)_{v\in\{\mathsf{G}\}\cup\mathcal{H}\setminus\{u\}}$ | $\mathbb{N}$ | Activation counters | 0 |

Table 2.3: Variables in $\mathsf{M}_u^*$

**end for**
**end transition**

**transition** $\mathsf{aut}_{v,u}?$ (initialized) *# Notification for other machines*
**enabled if:** $(state_u \neq \mathsf{corrupted}) \wedge (\mathsf{aut}_{v,u}?.cntr < tb)$;
$state_v \leftarrow \mathsf{init}$;
**output:** $\mathsf{out}_u!$ (initialized, $v$), $\mathsf{out}_u{}^{\triangleleft}!$ (1);
**end transition**

**transition** $\mathsf{in}_u?$ (new, $sid, grp$) *# Start new session*
**enabled if:** $(state_u \neq \mathsf{corrupted}) \wedge (\mathsf{in}_u?.cntr < tb)$;
**ignore if:**
$(u \notin grp) \vee (|grp| < 2) \vee (\exists v \in grp : state_v \neq \mathsf{init}) \vee (ses_{sid,grp} \neq \mathsf{undef})$;
$x_{sid,grp} \xleftarrow{\mathcal{R}} \mathbb{Z}_{|G|}$;
$ses_{sid,grp} \leftarrow \mathsf{upflow}$;
**if** $(u = grp[1])$ **then** *# $u$ is the first member*
$m'_1 \leftarrow g$;
$m'_2 \leftarrow g^{x_{sid,grp}}$
**output:** $\mathsf{aut}_{u,grp[2]}!$ (up, $sid$, $grp$, $(m'_1, m'_2)$);
$ses_{sid,grp} \leftarrow \mathsf{downflow}$;
**end if**
**end transition**

**transition** $\mathsf{aut}_{v,u}?$ (up, $sid, grp, msg$) *# Upflow message arrives*
**enabled if:** $(state_u \neq \mathsf{corrupted}) \wedge (\mathsf{aut}_{v,u}?.cntr < tb)$;
**ignore if:** $(ses_{sid,grp} \neq \mathsf{upflow}) \vee (v \neq grp[\mathsf{idx}(grp,u)-1]) \vee$
($msg$ is not a list $(m_1, \ldots, m_{\mathsf{idx}(grp,u)})$ over $G$ );
$i \leftarrow \mathsf{idx}(grp, u)$; *# $u$'s position in the group*
$m'_1 \leftarrow m_i$;
**for** $1 \leq j \leq \mathsf{min}(i, |grp| - 1)$ **do**
$m'_{j+1} \leftarrow m_j^{x_{sid,grp}}$
**end for**
**if** $(i < |grp|)$ **then**
**output:** $\mathsf{aut}_{u,grp[i+1]}!$ (up, $sid$, $grp$, $(m'_1, \ldots, m'_{i+1})$);
$ses_{sid,grp} \leftarrow \mathsf{downflow}$;
**else** *# $i = |grp|$, i.e., $u$ is the last member*
$key_{sid,grp} \leftarrow \mathsf{h}((m_{|grp|})^{x_{sid,grp}})$;

**if** $(ct = \mathsf{static})$ **then** # *For the static case we are done*
$ses_{sid,grp} \leftarrow \mathsf{finished}$;
**output:** $\mathsf{out}_u!$ ($\mathsf{key}$, $sid, grp, key_{sid,grp}$), $\mathsf{out}_u{}^{\triangleleft}!$ (1);
**else** # *For the adaptive case wait first for the confirmation flows*
$ses_{sid,grp} \leftarrow \mathsf{confirm}$;
$\mathcal{C}_{sid,grp} \leftarrow \{u\}$;
$x_{sid,grp} = \mathsf{undef}$; # *Erase secret exponent*
**end if**
**for all** $v' \in grp \setminus \{u\}$ **do** # *"Broadcast" to the group members*
**output:** $\mathsf{aut}_{u,v'}!$ ($\mathsf{down}$, $sid$, $grp$, $(m'_1, \ldots, m'_i)$);
**end for**
**end if**
**end transition**

**transition** $\mathsf{aut}_{v,u}?$ ($\mathsf{down}$, $sid, grp, msg$) # *Downflow message arrives*
**enabled if:** $(state_u \neq \mathsf{corrupted}) \wedge (\mathsf{aut}_{v,u}?.cntr < tb)$;
**ignore if:** $(ses_{sid,grp} \neq \mathsf{downflow}) \vee (v \neq grp[|grp|]) \vee$
($msg$ is not a list $(m_1, \ldots, m_{|grp|})$ over $G$ );
$i \leftarrow \mathsf{idx}(grp, u)$; # *u's position in the group*
$key_{sid,grp} \leftarrow \mathsf{h}((m_{|grp|+1-i})^{x_{sid,grp}})$;
**if** $(ct = \mathsf{static})$ **then** # *For the static case we are done*
$ses_{sid,grp} = \mathsf{finished}$;
**output:** $\mathsf{out}_u!$ ($\mathsf{key}$, $sid, grp, key_{sid,grp}$), $\mathsf{out}_u{}^{\triangleleft}!$ (1);
**else** # *For the adaptive case, start confirmation*
$ses_{sid,grp} \leftarrow \mathsf{confirm}$;
$\mathcal{C}_{sid,grp} \leftarrow \mathcal{C}_{sid,grp} \cup \{u, v\}$;
$x_{sid,grp} = \mathsf{undef}$; # *Erase secret exponent*
**for all** $v' \in grp \setminus \{u\}$ **do** # *"Broadcast" confirmation to group members*
**output:** $\mathsf{aut}_{u,v}!$ ($\mathsf{confirm}$, $sid, grp$);
**end for**
**if** $(\mathcal{C}_{sid,grp} = grp)$ **then** # *We got* $\mathsf{down}$ *after all* $\mathsf{confirm}$ *...*
$ses_{sid,grp} = \mathsf{finished}$; # *... so we are done: Give key to user ...*
**output:** $\mathsf{out}_u!$ ($\mathsf{key}$, $sid, grp, key_{sid,grp}$), $\mathsf{out}_u{}^{\triangleleft}!$ (1);
$key_{sid,grp} \leftarrow \mathsf{undef}$; # *... and delete it locally*
**end if**
**end if**

**end transition**

**transition** $\mathsf{aut}_{v,u}?$ $(\mathsf{confirm}, sid, grp)$ # *Confirmation message arrives*
  **enabled if:** $(ct = \mathsf{adaptive}) \wedge (state_u \neq \mathsf{corrupted}) \wedge (\mathsf{aut}_{v,u}?.cntr < tb)$;
  **ignore if:** $(v \notin grp \setminus \mathcal{C}_{sid,grp}) \vee (ses_{sid,grp} \notin \{\mathsf{downflow}, \mathsf{confirm}\})$;
  $\mathcal{C}_{sid,grp} \leftarrow \mathcal{C}_{sid,grp} \cup \{v\}$;
  **if** $(\mathcal{C}_{sid,grp} = grp) \wedge (ses_{sid,grp} = \mathsf{confirm})$ **then** # *All* confirm *received ...*
    $ses_{sid,grp} \leftarrow \mathsf{finished}$; # *... so we are done: Give key to user ...*
    **output:** $\mathsf{out}_u!$ $(\mathsf{key}, sid, grp, key_{sid,grp})$, $\mathsf{out}_u{}^{\triangleleft}!$ $(1)$;
    $key_{sid,grp} \leftarrow \mathsf{undef}$; # *... and delete it locally*
  **end if**
**end transition**

$\diamond$

The derivation of the actual system from the intended structure is now made as defined in Section 2.2.2. For example, the ports $\mathsf{aut}_{u,v}!$ are duplicated and passed to the adversary on port $\mathsf{aut}^{\mathsf{d}}_{u,v}!$. Similarly, a corruption switches a machine into transparent mode.

## *2.5 Security of Real System*

**Theorem 2.1 (Security of Scheme 2.2)** *For all* $n \in \mathbb{N}$ *and* $ct \in \{\mathsf{static}, \mathsf{adaptive}\}$

$$Sys^{\mathsf{gke,ika1}}_{n,tb,ct} \geq_{\mathsf{sec}} Sys^{\mathsf{gke,ideal}}_{n,tb,ct}$$

$\square$

We prove Theorem 2.1 in several steps:

First, we define an interactive version of the $n$-party Diffie-Hellman decision problem, and show that it is hard provided the ordinary Diffie-Hellman decision problem is hard. We do this by defining two (computationally indistinguishable) machines, $\mathsf{GDH}^{(0)}_{n,mxkey}$ and $\mathsf{GDH}^{(1)}_{n,mxkey}$, where $n$ denotes the maximum size of a group and *mxkey* the overall maximum number of key agreements. The former computes keys as in the real protocol while the latter is idealized: It works like $\mathsf{GDH}^{(0)}_{n,mxkey}$, but instead of producing the correct key as $\mathsf{h}(g^{x_1 \cdots x_n})$ it produces some random bit string of the appropriate length.

Next, we rewrite the real system such that all partial Diffie-Hellman keys of all machines $\mathsf{M}_u$ are computed by a hypothetical joint submachine

$\mathsf{GDH}^{(0)}_{n,mxkey}$. Thus, we separate the computational indistinguishability aspects from others like state keeping (e.g., to show the sufficiency of confirmation messages in handling adaptive adversaries.) By the composition theorem from [2], we can replace this submachine by $\mathsf{GDH}^{(1)}_{n,mxkey}$.

Finally, we show that the resulting system is perfectly indistinguishable from the trusted host together with a suitable simulator.

### 2.5.1 Generalized Diffie-Hellman Machine

Let us look first at the underlying number-theoretic assumptions and two auxiliary lemmas required in proving the main theorem of this section. The Decisional Generalized Diffie-Hellman Problem (DGDH($n$)) was introduced by [18] and is a natural extension of the 2-party Decisional Diffie-Hellman assumption (DDH) to an $n$-party case: Given all partial DH-keys $\{g^{\prod_{\beta_i=1} x_i} \mid \beta \in I_n \setminus \{1^n\}\}$ and a value $g^c$, the task is to decide whether $g^c$ is $g^{\prod x_i}$ or a random element of $G$. As the group elements in the flows of DH-based group-key agreements form a subset of the partial keys, DGDH($n$) is the natural assumption to base the security on. However, DGDH($n$) is not a standard assumption. Preferably, we could rely on a standard assumption such as DGDH($n$). DGDH($n$) is used in many contexts [41] and assumed to hold for many cyclic groups, e.g., Shoup [40] showed that no polynomial algorithm can solve DDH in the generic model if the group order contains only large prime factors. Luckily, with the following Lemma 2.1 we can equate the two assumptions. The lemma is taken from [18], adapted and generalized to the classification and notation of [42][13] and extended with the concrete security of the reduction:[14]

**Lemma 2.1 (Decisional Generalized Diffie-Hellman Problem)**

[13]For our context it is enough to know that $Y$-$X$ (c:u,g:l,f:nsprim) denotes the assumption that there are no non-negligibly ($Y = 1/\mathsf{poly}(k)$) or overwhelmingly ($Y = (1-1/\mathsf{poly}(k))$) successful oracles breaking the problem $X$ in uniform complexity (c:u), low granularity (g:l), and in a group where the order does not contain any small prime factors (f:nsprim).

[14]This is denoted by $A \xrightarrow{t' \leq f_t(t,\alpha,\ldots);\ \alpha' \geq f_\alpha(t,\alpha,\ldots)} B$ and means that if assumption $B$ can be broken in time $t$ and with success probability $\alpha$ we can break $A$ in time $t'$ and with success probability $\alpha'$ bounded by functions $f_t$ and $f_\alpha$, respectively.

$1/\mathsf{poly}(k)$-DDH(c:u; g:l; f:nsprim,o)
$$\xrightarrow{\alpha'=\alpha/O(n);\ t'=t+O(2^n \log(|G|))}$$
$1/\mathsf{poly}(k)$-DGDH($n$)(c:u; g:l; f:nsprim,o)
□

The proof of the lemma can be found in [43] or [44].

*Remark 2.1.* The factor $2^n$ in the reduction cost gives a pretty bad efficiency but is unavoidable due to the size of a GDH(n) instance. However, in practice the number $\#pkey$ of partial keys visible to an adversary is small (usually, $O(n^2)$ in group key agreement protocols). By suitably ignoring partial keys which are not in the adversary's view, we can improve to a time complexity of at most $t + O(n\ \#pkey\ \log(|G|))$ with the same success probability. ∘

We can weaken the assumption even further by requiring only the nonexistence of oracles which solve virtually all problem instances. The following Lemma is very useful in achieving this goal.

**Lemma 2.2**

$(1-1/\mathsf{poly}(k))$-DDH(c:u; g:m; f:nsprim,o)
$$\xrightarrow{\alpha'\geq 1-1/2^k;\ t'=(O(n^2k/\alpha^2)(t+O(2^n \log(|G|))))}$$
$1/\mathsf{poly}(k)$-DGDH($n$)(c:u; g:l; f:nsprim,o)
□

*Proof.* The assumption $(1-1/\mathsf{poly}(k))$-DDH(c:u; g:m; f:nsprim,o) implies $1/\mathsf{poly}(k)$-DDH(c:u; g:m; f:nsprim,o) due to the existence of a self-corrector for DDH [45, 46]. From the fact that a medium granularity assumption implies a low granularity assumption [42] it follows that $1/\mathsf{poly}(k)$-DDH(c:u; g:m; f:nsprim,o) implies $1/\mathsf{poly}(k)$-DDH(c:u; g:l; f:nsprim,o) . Finally, it follows from Lemma 2.1 that the $1/\mathsf{poly}(k)$-DDH(c:u; g:l; f:nsprim,o) assumption implies $1/\mathsf{poly}(k)$-DGDH($n$)(c:u; g:l; f:nsprim,o) . From this the lemma follows by transitivity of $\Longrightarrow$ . ■

*Remark 2.2.* As a curious reader you might wonder why we did not base the security on the weaker low-granularity assumption $(1-1/\mathsf{poly}(k))$-DDH(c:u; g:l; f:nsprim,o) . Unfortunately (and opposite to

what is implicitly claimed by [41]), the self-correctors for DDH from [45] or [46] do not work for low granularity as the "classical" random self-reducibility does not apply to the low granularity case and no other approach of amplifying the oracle is known so far. ∘

Combining this result with the proof of Shoup [40] that DDH is provably hard in the generic model trivially leads to following result:

**Corollary 2.1** *true* $\Longrightarrow$ $(1-1/\mathsf{poly}(k))$-DGDH$(n)^{\sigma}$(c:∗; g:m; f:nsprim) □

This gives us confidence, that under suitable choice of algebraic group, namely that the group order does not contain any small primes, this is a good assumption to base the security of a protocol upon.

The ideal host requires a random $k$-bit string as key. Therefore, we cannot use the Diffie-Hellman key directly and we have to derive a random bit-string from it using a (random) universal hash function $\mathsf{h}$ from $G$ to $\{0,1\}^k$. The security of this approach is shown in following lemma:

**Lemma 2.3 (Key Derivation)**

$$1/\mathsf{poly}(k)\text{-DGDH}(n)(\text{c:u; g:l; f:nsprim,o}) \ \wedge \ |G| \geq 2^{3k}$$
$$\xRightarrow{\alpha' \geq \alpha - 2^{-k};\ t'=t}$$
$$(\mathsf{h}(g^{\prod x_i}), GDH_{k,n}^{Public}(x_1,\ldots,x_n), \overline{\mathsf{h}}) \overset{c}{\approx} (K, GDH_{k,n}^{Public}(x_1,\ldots,x_n), \overline{\mathsf{h}})$$

*where* $\mathsf{h} \in_{\mathcal{R}} UHF_{G,k}$, $\overline{\mathsf{h}}$ *denotes a description of the function* $\mathsf{h}$, $GDH_{k,n}^{Public}(x_1,\ldots,x_n)$ *is the publicly known part of a DGDH tuple with exponents* $x_1,\ldots,x_n$ *randomly chosen from* $\mathbb{Z}_{|G|}^n$, *and* $K \in_{\mathcal{R}} \{0,1\}^k$. □

The proof of the lemma can be found in [43] or [44].

As mentioned in the introduction of this section, our goal is to abstract the computation of keys and, indirectly, the underlying number-theoretic problem in a clean way. This is achieved with the following machine and its two modes of operation determined by the parameter $b$:

**Scheme 2.3 (Generalized Diffie-Hellman Machine $\mathsf{GDH}_{n,mxkey}^{(b)}$)**

The machines $\mathsf{GDH}_{n,mxkey}^{(b)}$, for $b \in \{0,1\}$, are constructed as follows: $n$ is the maximum number of members in any session, *mxkey* is the maximum number of sessions. $\mathsf{GDH}_{n,mxkey}^{(b)}$ has ports $\{\mathsf{in}_{\mathsf{gdh}}?, \mathsf{out}_{\mathsf{gdh}}!, \mathsf{out}_{\mathsf{gdh}}{}^{\triangleleft}!\}$, where in each transition triggered at $\mathsf{in}_{\mathsf{gdh}}?$ exactly one output is sent to $\mathsf{out}_{\mathsf{gdh}}!$ which is

| Port | Type | Parameters | Meaning |
|---|---|---|---|
| $\mathsf{in}_{\mathsf{gdh}}?$ | init | — | Get system parameters |
| $\mathsf{out}_{\mathsf{gdh}}!$ | initR | $G, g, \mathsf{h}$ | Reply to above |
| $\mathsf{in}_{\mathsf{gdh}}?$ | getView | $n'$ | Get GDH partial keys of a new session |
| $\mathsf{out}_{\mathsf{gdh}}!$ | getViewR | $i, \{(\beta, g^{\prod_{\beta_j=1} x_{i,j}}) \mid \beta \in I_{n_i} \setminus \{1^{n_i}\}\}$ | Reply to above, $i$ is the session reference identifier |
| $\mathsf{in}_{\mathsf{gdh}}?$ | getKey | $i$ | Get key of session $i$ |
| $\mathsf{out}_{\mathsf{gdh}}!$ | getKeyR | $z_i$ | Reply to above |
| $\mathsf{in}_{\mathsf{gdh}}?$ | getSecret | $i$ | Get secret exponents of session $i$ |
| $\mathsf{out}_{\mathsf{gdh}}!$ | getSecretR | $(x_{i,1}, \ldots, x_{i,n_i})$ | Reply to above |

Table 2.4: The message types and parameters handled by $\mathsf{GDH}^{(b)}_{n,mxkey}$.

immediately scheduled. As a convention we will call such self-clocked request-reply pairs **remote procedure calls (RPC)** and replies to message type mt will always have message type mtR.

A machine $\mathsf{GDH}^{(b)}_{n,mxkey}$ handles the messages shown in Table 2.4 and contains the variables shown in Table 2.5. The state transition functions are defined in following rules:

**transition** $\mathsf{in}_{\mathsf{gdh}}?$ (init)
  **enabled if:** $(i = 0)$;
  $(G, g, \mathsf{h}) \xleftarrow{\mathcal{R}} \mathsf{genG}(1^k)$;
  $i \leftarrow 1$;
  **output:** $\mathsf{out}_{\mathsf{gdh}}!$ (initR, $G, g, \mathsf{h}$), $\mathsf{out}_{\mathsf{gdh}}{}^{\triangleleft}!$ (1);
**end transition**

**transition** $\mathsf{in}_{\mathsf{gdh}}?$ (getView, $n'$)
  **enabled if:** $(1 \leq i \leq mxkey)$; # *Initialized & maxima not exceeded*
  **ignore if:** $\neg(2 \leq n' \leq n)$; # *Illegal number of participants*
  $c_i \leftarrow$ init;
  $n_i \leftarrow n'$;

| Name | Domain | Meaning | Init. |
|---|---|---|---|
| $(G, g, \mathsf{h})$ | Range of $\mathsf{genG}(1^k)$ | System parameters | |
| $i$ | $\mathbb{N}$ | Session counter | 0 |
| $(c_i)_{i\in\mathbb{N}}$ | {undef, init, finished, corrupted } | Session status | undef |
| $(n_i)_{i\in\mathbb{N}}$ | $\mathbb{N}$ | Number of session participants | |
| $(x_{i,j})_{i,j\in\mathbb{N}}$ | $\mathbb{Z}_{\lvert G\rvert}$ | Secret exponents | |
| $(z_i)_{i\in\mathbb{N}}$ | $G$ | Session keys | |
| $\mathsf{in}_{\mathsf{gdh}}?.cntr$ | $\mathbb{N}$ | Activation counter | 0 |

Table 2.5: Variables in $\mathsf{GDH}^{(b)}_{n,mxkey}$

$(x_{i,1}, \ldots, x_{i,n_i}) \xleftarrow{\mathcal{R}} \mathbb{Z}^{n_i}_{|G|}$;
**if** $b = 0$ **then** # *Depending on type of machine ...*
$z_i \leftarrow \mathsf{h}(g^{x_{i,1}\cdots x_{i,n_i}})$; # *... set real key ...*
**else**
$z_i \xleftarrow{\mathcal{R}} \{0,1\}^k$; # *... or random key.*
**end if**
**output:** $\mathsf{out}_{\mathsf{gdh}}!$ (getViewR, $i, \{(\beta, g^{\prod_{\beta_j=1} x_{i,j}}) \mid \beta \in I_{n_i} \backslash \{1^{n_i}\}\}$), $\mathsf{out}_{\mathsf{gdh}}{}^{\triangleleft}!$ (1);
$i \leftarrow i + 1$;
**end transition**

**transition** $\mathsf{in}_{\mathsf{gdh}}?$ (getKey, $i$)
**ignore if:** $(c_i \neq \mathsf{init})$; # *Session not yet initialized or already terminated*
$c_i \leftarrow \mathsf{finished}$;
**output:** $\mathsf{out}_{\mathsf{gdh}}!$ (getKeyR, $z_i$), $\mathsf{out}_{\mathsf{gdh}}{}^{\triangleleft}!$ (1);
**end transition**

**transition** $\mathsf{in}_{\mathsf{gdh}}?$ (getSecret, $i$)
**ignore if:** $(c_i \neq \mathsf{init})$; # *Session not yet initialized or already terminated*
$c_i \leftarrow \mathsf{corrupted}$;
**output:** $\mathsf{out}_{\mathsf{gdh}}!$ (getSecretR, $(x_{i,1}, \ldots, x_{i,n_i})$), $\mathsf{out}_{\mathsf{gdh}}{}^{\triangleleft}!$ (1);
**end transition**

$\diamond$

Let me briefly motivate the transitions. The meaning of the init message should be clear: It causes the initialization of the machine and the generation of the system parameters. Using a getView message, a caller can then instantiate a particular instance of a GDH problem and retrieve all corresponding partial GDH keys. We will use this later to generate the messages exchanged in a session of the key establishment protocol. The purpose of getKey is to provide a key corresponding to the partial GDH keys returned by getView. Depending on the bit $b$, this will result in the correctly derived key or an independent random bit-string of the appropriate length, respectively. Therefore, we can satisfy our goal of decoupling the actual session key from the messages in a key establishment session by setting $b = 1$. However, in sessions with dishonest group members, e.g., due to a corruption, this strategy will not work. In these cases, the protocol messages might contain elements of the group $G$ other than the partial GDH keys. Even worse, we also cannot use the "fake" session key provided by getKey. The dishonest members, i.e., the adversary, can correctly derive the "real" session key from the GDH partial keys and the secret exponents. Therefore, the adversary would immediately detect the difference. This explains the existence of getSecret. It provides us with all secret exponents and allows us to also handle corrupted sessions. Finally, note that for each session only either getSecret or getKey can be called successfully!

As we will show in the following lemma, views from the two machines $\mathsf{GDH}^{(b)}_{n,mxkey}$ are indistinguishable if the DGDH($n$) assumption (and indirectly the DDH assumption) holds. Note that this does not immediately follow from the DGDH($n$) assumption: The interactivity, in particular corruptions (modeled by calls to getSecret), requires special attention.

**Lemma 2.4** For any $n \geq 2$ and *mxkey* and any polynomial-time machine A it holds that

$$(1-1/\mathsf{poly}(k))\text{-DDH(c:u; g:m; f:nsprim,o)} \xrightarrow{\alpha' \geq 1-1/2^k;\ t' \leq (t+O(mxkey\ 2^n k^3))(O(n^2k)/\alpha^2)} view^{(0)} \overset{c}{\approx} view^{(1)}$$

where $view^{(b)}$ denotes the view of A while interacting with $\mathsf{GDH}^{(b)}_{n,mxkey}$. □

*Proof.* Assume that there is an interactive machine $\mathsf{D_A}$ which can distinguish $view^{(0)}$ from $view^{(1)}$ with non-negligible advantage $\delta :=$

$|\mathbf{Prob}[\mathsf{D}_\mathsf{A}(view^{(b)}) = b :: b \xleftarrow{\mathcal{R}} \{0,1\}] - 0.5|$.

Without loss of generality, we can assume that $\mathsf{A}$ always uses $n' = n$: We can always transform outputs for $n$ into outputs for an $n' < n$ by virtually combining $x_{n'}, x_{n'+1}, \ldots, x_n$ into a single value $\prod_{j=n'}^{n} x_j$, i.e., we delete from $\{(\beta, g^{\prod_{\beta_j=1} x_{i,j}} \mid \beta \in I_{n'} \setminus \{1^{n'}\})$ all pairs where not all values $\beta_j$ for $j = n', \ldots, n$ are equal, and for the remaining ones we replace $\beta$ by $\beta_1 \ldots \beta_{n'}$. In the output generated on input $\mathsf{getSecret}$, we replace $x_{n'}$ by $\prod_{j=n'}^{n} x_j$ and omit all $x_i$ with $i > n'$. It is easy to see that everything is consistent and correctly distributed ($\prod_{j=n'}^{n} x_j$ is statistically indistinguishable from a uniformly chosen $x \in \mathbb{Z}_{|G|}$.) Now the lemma follows from a *hybrid argument*: Let us define $mxkey + 1$ hybrid machines $\mathsf{GDH}^{\{i\}}_{n,mxkey}$. The machine $\mathsf{GDH}^{\{i\}}_{n,mxkey}$ is built and behaves like $\mathsf{GDH}^{(1)}_{n,mxkey}$ but flips the bit $\mathsf{GDH}^{\{i\}}_{n,mxkey}.b$ to 0 before handling the $i$-th $\mathsf{getView}$ request. Clearly, the extreme hybrids $\mathsf{GDH}^{\{1\}}_{n,mxkey}$ and $\mathsf{GDH}^{\{mxkey+1\}}_{n,mxkey}$ are identical to $\mathsf{GDH}^{(0)}_{n,mxkey}$ and $\mathsf{GDH}^{(1)}_{n,mxkey}$, respectively. Let $\delta_i$ be $\mathsf{D}_\mathsf{A}$'s advantage of distinguishing $\mathsf{GDH}^{\{i\}}_{n,mxkey}$ from $\mathsf{GDH}^{\{i+1\}}_{n,mxkey}$.

Using $\mathsf{A}$ and $\mathsf{D}_\mathsf{A}$ as a subroutine we can now construct a distinguisher $\mathsf{D}$ which distinguishes $GDH^{(0)}_{k,n}$ from $GDH^{(1)}_{k,n}$: Given a sample $GDH_{k,n} \leftarrow GDH^{(b)}_{k,n}$, $\mathsf{D}$ first picks $c \xleftarrow{\mathcal{R}} \{1, \ldots, mxkey\}$. Then it starts and interacts with $\mathsf{A}$ behaving like $\mathsf{GDH}^{\{c\}}_{n,mxkey}$ with the following exceptions:[15] When it receives an $\mathsf{init}$ query, it replaces $G$ and $g$ returned by $\mathsf{genG}(1^k)$ with the group and generator associated with $GDH_{k,n}$; in the $c$-th $\mathsf{getView}$ query it answers with $(\mathsf{getViewR}, c, GDH^{\mathrm{Public}}_{k,n})$; on valid (i.e., $c_c \neq \mathsf{init}$) input $(\mathsf{getKey}, c)$ it returns $(\mathsf{getKeyR}, c, \mathsf{h}(GDH^{\mathrm{Key}}_{k,n}))$; and on valid input $(\mathsf{getSecret}, c)$ it simply gives up (it cannot correctly answer that request), outputs bit $b' \xleftarrow{\mathcal{R}} \{0,1\}$ and halts. Finally, when $\mathsf{A}$ terminates with view $view_\mathsf{A}$ it outputs $b' \leftarrow \mathsf{D}_\mathsf{A}(view_\mathsf{A})$ and halts.

Let $\mathsf{D}^{\{i\}}$ denote $\mathsf{D}$ with $c$ chosen as $i$. Further, let $\mathsf{bad}_i$ be the event that a valid input $(\mathsf{getSecret}, i)$ occured, i.e., the event which makes $\mathsf{D}^{\{i\}}$ give up. Note that the distribution of $G$, $g$, $\mathsf{h}$ and exponents of DGDH-tuples produced by $\mathsf{D}^{\{i\}}$ is identical to the equivalent distribution in $GDH^{(b)}_{k,n}$ due to the well-behavior of $\mathsf{genG}$. Therefore, if $\mathsf{bad}_i$ does not happen then $\mathsf{D}^{\{i\}}$ behaves exactly like $\mathsf{A}$ interacting with $\mathsf{GDH}^{\{c+b\}}_{n,mxkey}$.

[15] Note that the changes apply only for cases where the **require:** condition is fulfilled, otherwise the requests are rejected as usual.

Let the probability of $\mathsf{D}$ in guessing $b$ correctly be written as

$$\begin{aligned}&\mathbf{Prob}[b' = b] = \\ &\textstyle\sum_{i=1}^{mxkey} \mathbf{Prob}[c = i]\ \ (\mathbf{Prob}[b' = b | \mathsf{bad}_i \wedge c = i]\mathbf{Prob}[\mathsf{bad}_i] + \\ &\qquad\qquad \mathbf{Prob}[b' = b | \neg\mathsf{bad}_i \wedge c = i]\mathbf{Prob}[\neg\mathsf{bad}_i]).\end{aligned}$$

As $\mathsf{D}^{\{i\}}$ simulates $\mathsf{A}$'s environment perfectly up to a possible occurrence of $\mathsf{bad}_i$, the probability of $\mathsf{bad}_i$ is the same for $\mathsf{D}^{\{i\}}$ as for views of $\mathsf{A}$ when operating in reality. Additionally, views of $\mathsf{A}$ from the $i$-th and the $i+1$-th hybrids conditioned on the occurrence of $\mathsf{bad}_i$ are identical in reality (without giving up) because the only difference, $z_i$, is not output. So $\mathsf{D_A}$ has to guess (as does $\mathsf{D}^{\{i\}}$), i.e.,

$$\mathbf{Prob}[\mathsf{D_A}(view_{\mathsf{GDH}^{\{i+b\}}_{n,mxkey}}) = b | \mathsf{bad}_i] = \mathbf{Prob}[\mathsf{D}^{\{i\}}(GDH_{k,n}) = b | \mathsf{bad}_i] = 0.5.$$

If $\mathsf{bad}_i$ does not occur, then $\mathsf{D}^{\{i\}}$ perfectly simulates $\mathsf{GDH}^{\{i+b\}}_{n,mxkey}$ so

$$\mathbf{Prob}[\mathsf{D_A}(view_{\mathsf{GDH}^{\{i+b\}}_{n,mxkey}}) = b | \neg\mathsf{bad}_i] = \mathbf{Prob}[\mathsf{D}^{\{i\}}(GDH_{k,n}) = b | \neg\mathsf{bad}_i].$$

By combining the previous two equations it follows that

$$\mathbf{Prob}[\mathsf{D_A}(view_{\mathsf{GDH}^{\{i+b\}}_{n,mxkey}}) = b] = \mathbf{Prob}[\mathsf{D}^{\{i\}}(GDH_{k,n}) = b] = 0.5 + \delta_i$$

and by this and the first equation it has to hold that

$$\mathbf{Prob}[b' = b] = 1/2 + \frac{1}{mxkey} \sum_{i=1}^{mxkey} \delta_i.$$

Using the inequality $\sum_{i=1}^{mxkey} \delta_i \geq \delta$ and the hypothesis that the advantage $\delta$ of $\mathsf{D_A}$ is non-negligible, leads to an immediate contradiction of the $1/\mathsf{poly}(k)$-DGDH$(n)$(c:u; g:l; f:nsprim,o) assumption. The lemma then follows immediately from this contradiction and the Lemmas 2.2 and 2.1. ∎

### 2.5.2 Real System Rewritten with Interactive Diffie-Hellman Machine

We now rewrite the real system so that it uses $\mathsf{GDH}^{(0)}_{n,mxkey}$. We do this via a multiplexer GDH_Mux which maps group names, indices $u$, etc., of the individual modified machines $\mathsf{M}'_u$ to the simple sequence numbers of $\mathsf{GDH}^{(0)}_{n,mxkey}$,

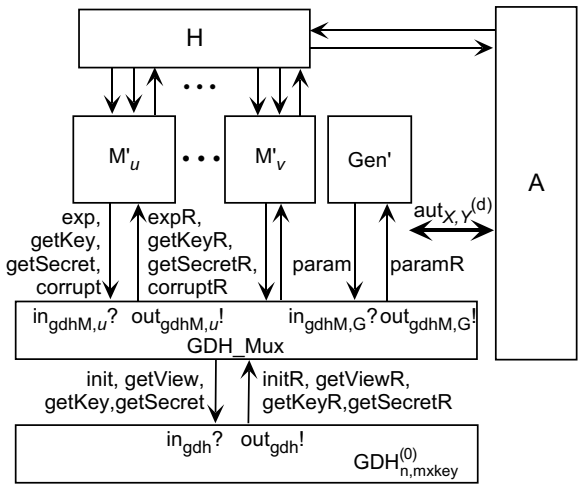


Figure 2.5: Semi-real system. (Clocking of new components GDH_Mux and $\mathsf{GDH}^{(0)}_{n,mxkey}$ is RPC-style.)

and distributes the parts of views to the machines as they need them. Essentially, this rewriting shows that the real system only uses Diffie-Hellman keys in the proper way captured in $\mathsf{GDH}^{(0)}_{n,mxkey}$, i.e., never outputting both a key and a secret, and that active attacks (where the machines raise adversary-chosen elements to secret powers) do not make a difference. The situation is summarized in Figure 2.5. More precisely, the system is defined as follows:

**Scheme 2.4 (Semi-real system $Sys^{\mathsf{gke,ika1,sr}}_{n,tb,ct}$)**

The structures of the semi-real system $Sys^{\mathsf{gke,ika1,sr}}_{n,tb,ct}$ contain machines $\mathsf{M}'_u$ for all $u \in \mathcal{H}$, $\mathsf{Gen}'$, GDH_Mux, and $\mathsf{GDH}^{(0)}_{n,mxkey}$, where $mxkey$ can be upper bounded according to the runtime of $\mathsf{M}'_u$, i.e., $tb$, as $n * tb$.[16]

$\mathsf{M}'_u$ and $\mathsf{Gen}'$ are identical to the corresponding $\mathsf{M}_u$ and Gen from scheme $Sys^{\mathsf{gke,ika1}}_{n,tb,ct}$ except that all operations on Diffie-Hellman keys are offloaded to GDH_Mux (see Figure 2.7 for the message interface of GDH_Mux towards these machines). $\mathsf{Gen}'$ gets additional ports $\{\mathsf{in}_{\mathsf{gdhM,G}}!, \mathsf{in}_{\mathsf{gdhM,G}}{}^{\triangleleft}!, \mathsf{out}_{\mathsf{gdhM,G}}?\}$. It uses them to get the system parameters by replacing the call to genG with

[16]This bound is of course overly conservative in practice. To get a considerably improved concrete security without much loss of generality, one could parameterize the model with additional bounds on the number of new requests and on the maximum size of a group. The changes throughout the model and proof would be cumbersome yet straightforward.

| Elementary action | Replaced by |
|---|---|
| $x_{sid,grp} \xleftarrow{\mathcal{R}} \mathbb{Z}_{\|G\|}$ | $x_{sid,grp} \leftarrow \mathsf{exists}$. |
| $m^* \leftarrow m^{x_{sid,grp}}$ | Output $\mathsf{in}_{\mathsf{gdhM},u}!$ ($\mathsf{exp}$, $sid$, $grp$, $m$) and use the answer as $m^*$. |
| $key_{sid,grp} \leftarrow \mathsf{h}(m^{x_{sid,grp}})$ | $key_{sid,grp} \leftarrow m$, i.e., delay key computation. |
| Output $key_{sid,grp}$ (when passing $\mathsf{key}$ to $\mathsf{H}$) | Output $\mathsf{in}_{\mathsf{gdhM},u}!$ ($\mathsf{getKey}$, $sid$, $grp$, $key_{sid,grp}$), i.e., perform delayed key computation, and use the answer as $key_{sid,grp}$. |
| Output $key_{sid,grp}$ (during corruption) | If $key_{sid,grp} \neq \mathsf{undef}$ (key computed but not yet erased) output $\mathsf{in}_{\mathsf{gdhM},u}!$ ($\mathsf{getKey}$, $sid$, $grp$, $key_{sid,grp}$) and use the answer as $key_{sid,grp}$. |
| Output $x_{sid,grp}$ (during corruption) | If $x_{sid,grp} = \mathsf{exists}$ (secret generated but not yet erased) output $\mathsf{in}_{\mathsf{gdhM},u}!$ ($\mathsf{getSecret}$, $sid$, $grp$) and use the answer as $x_{sid,grp}$. |

Table 2.6: Changed elementary actions in the semi-real machines $\mathsf{M}'_u$

a remote procedure call to $\mathsf{param}$ at $\mathsf{GDH\_Mux}$. $\mathsf{M}'_u$ has the same variables as $\mathsf{M}_u$. They also have the same meaning except that the domain of $\mathsf{M}'_u.x_{sid,grp}$ is extended with a distinct value $\mathsf{exists}$ and the domain of $\mathsf{M}'_u.key_{sid,grp}$ by $G$. $\mathsf{M}'_u$ has additional ports $\{\mathsf{in}_{\mathsf{gdhM},u}!, \mathsf{in}_{\mathsf{gdhM},u}{}^{\triangleleft}!, \mathsf{out}_{\mathsf{gdhM},u}?\}$ to communicate with $\mathsf{GDH\_Mux}$ via remote procedure calls. The cryptographic actions are changed as defined in Table 2.6. Additionally, on input $\mathsf{corrupt}_u?$ ($\mathsf{do}$), $\mathsf{M}'_u$ first outputs $\mathsf{in}_{\mathsf{gdhM},u}!$ ($\mathsf{corrupt}$) and waits for the response $\mathsf{corruptR}$. (And after the corruption, the forwarding only refers to the original ports of $\mathsf{M}_u$.)

$\mathsf{GDH\_Mux}$ has ports $\{\mathsf{in}_{\mathsf{gdh}}!, \mathsf{out}_{\mathsf{gdh}}?, \mathsf{in}_{\mathsf{gdh}}{}^{\triangleleft}!\} \cup \{\mathsf{in}_{\mathsf{gdhM},u}?, \mathsf{out}_{\mathsf{gdhM},u}!, \mathsf{out}_{\mathsf{gdhM},u}{}^{\triangleleft}! \mid u \in \mathcal{M} \cup \{\mathsf{G}\}\}$. At its "upper" ports, it handles the message types shown in Table 2.7. All of them are of the remote procedure call type, i.e., responses are immediately scheduled. The $\mathsf{GDH\_Mux}$ de-multiplexes requests to and from $\mathsf{GDH}^{(0)}_{n,mxkey}$ and shields $\mathsf{GDH}^{(0)}_{n,mxkey}$ from illegal requests, i.e., $\mathsf{GDH}^{(0)}_{n,mxkey}$ is asked at most one of $\mathsf{getSecret}$ and $\mathsf{getKey}$ for a given session, and handles corruptions. In the **require:**-clauses we collect the preconditions under which $\mathsf{GDH\_Mux}$ will get the desired correct answers from $\mathsf{GDH}^{(0)}_{n,mxkey}$; we will show below that they are always fulfilled in the overall

| Port | Type | Parameters | Meaning |
|---|---|---|---|
| $\mathsf{in}_{\mathsf{gdhM,G}}?$ | param | — | Get system parameters |
| $\mathsf{out}_{\mathsf{gdhM,G}}!$ | paramR | $G$, $g$, $\mathsf{h}$ | Reply to above |
| $\mathsf{in}_{\mathsf{gdhM},u}?$ | corrupt | — | Corruption |
| $\mathsf{out}_{\mathsf{gdhM},u}!$ | corruptR | — | Reply to above |
| $\mathsf{in}_{\mathsf{gdhM},u}?$ | exp | $sid$, $grp$, $\gamma$ | Exponentiate $\gamma$ with secret for $u$ in this session. Limited to the computation of partial keys! |
| $\mathsf{out}_{\mathsf{gdhM},u}!$ | expR | $\gamma^{x_u}$ | Reply to above |
| $\mathsf{in}_{\mathsf{gdhM},u}?$ | getKey | $sid$, $grp$, $\gamma$ | Get derived key matching final partial key $\gamma$ |
| $\mathsf{out}_{\mathsf{gdhM},u}!$ | getKeyR | $K$ | Reply to above |
| $\mathsf{in}_{\mathsf{gdhM},u}?$ | getSecret | $sid$, $grp$ | Get secret of this session (to hand it over during corruption) |
| $\mathsf{out}_{\mathsf{gdhM},u}!$ | getSecretR | $x_u$ | Reply to above |

Table 2.7: Messages at "Upper" ports of GDH_Mux where $u$ ranges over $\mathcal{M}$.

semi-real system.

The variables of GDH_Mux are shown in Table 2.8. Below follows the state transition functions of GDH_Mux. Note that requests to $\mathsf{in}_{\mathsf{gdh}}?$ are remote procedure calls immediately answered by $\mathsf{GDH}^{(b)}_{n,mxkey}$. Therefore, we do not define special wait-states where GDH_Mux waits for these answers, but treat them within the surrounding transitions. We further assume that the corresponding input port $\mathsf{out}_{\mathsf{gdh}}?$ is enabled only for a single outstanding reply. For an $n$-bit string $\beta$ and $1 \leq i \leq n$, let $\mathsf{bit}(\beta, i)$ denote the $i$-th bit in $\beta$ and $\mathsf{setbit}(\beta, i)$ denote that the $i$-th bit in $\beta$ is set to one. Furthermore, let "$\beta :: \mathsf{predicate}(\beta)$" means "all $\beta$ such that predicate predicate holds".

**transition** $\mathsf{in}_{\mathsf{gdhM,G}}?$ (param)
  **output:** $\mathsf{in}_{\mathsf{gdh}}!$ (init), $\mathsf{in}_{\mathsf{gdh}}{}^{\triangleleft}!$ (1);
  **input:** $\mathsf{out}_{\mathsf{gdh}}?$ (initR, $G, g, \mathsf{h}$);
  **output:** $\mathsf{out}_{\mathsf{gdhM,G}}!$ (paramR, $G, g, \mathsf{h}$), $\mathsf{out}_{\mathsf{gdhM,G}}{}^{\triangleleft}!$ (1);
**end transition**

| Variables | Domain | Meaning | Init. |
|---|---|---|---|
| $(i_{sid,grp})_{sid \in \mathcal{SID}, grp \subseteq \mathcal{M}}$ | $\mathbb{N}$ | Index used for this session with $\mathsf{GDH}^{(b)}_{n,mxkey}$ | undef |
| $(corr_u)_{u \in \mathcal{M}}$ | $\{\mathsf{true}, \mathsf{false}\}$ | Corrupted machine? | true iff $u \in \mathcal{M} \setminus \mathcal{H}$ |
| $(ses_{u,sid,grp})_{u \in \mathcal{M}, sid \in \mathcal{SID}, grp \subseteq \mathcal{M}}$ | $\{\mathsf{undef}, \mathsf{finished}, \mathsf{corrupted}\}$ | Session status related to $u$ | undef |
| $(key_{sid,grp})_{sid \in \mathcal{SID}, grp \subseteq \mathcal{M}}$ | $\{0,1\}^k \cup \{\mathsf{undef}\}$ | Session key from $\mathsf{GDH}^{(b)}_{n,mxkey}$ | undef |
| $(view_{sid,grp})_{sid \in \mathcal{SID}, grp \subseteq \mathcal{M}}$ | As output by $\mathsf{GDH}^{(b)}_{n,mxkey}$ | View of a session | undef |
| $(secrets_{sid,grp})_{sid \in \mathcal{SID}, grp \subseteq \mathcal{M}}$ | As output by $\mathsf{GDH}^{(b)}_{n,mxkey}$ | Secrets of a session | undef |
| $(\mathsf{in}_{\mathsf{gdhM},u}?.cntr)_{u \in \in \mathcal{M} \cup \{\mathsf{G}\}}$ $\mathsf{out}_{\mathsf{gdh}}?.cntr$ | $\mathbb{N}$ | Activation counters | 0 |

Table 2.8: Variables in GDH_Mux

**transition** $\mathsf{in}_{\mathsf{gdhM},u}?$ (exp, $sid, grp, \gamma$)

**require:** $(u \in grp) \ \wedge \ ((i_{sid,grp} = \mathsf{undef})$
$\vee \ ((\exists v \in grp : ses_{v,sid,grp} = \mathsf{corrupted}) \wedge (key_{sid,grp} = \mathsf{undef}))$
$\vee \ ((\forall v \in grp : ses_{v,sid,grp} \neq \mathsf{corrupted}) \wedge (\exists \beta : (\beta, \gamma) \in view_{sid,grp} \wedge \mathsf{bit}(\beta,$
$\mathsf{idx}(grp, u)) = 0 \wedge \mathsf{setbit}(\beta, \mathsf{idx}(grp, u)) \neq 1^{|grp|})))$;
*# A legitimate caller and either session is completely undefined or ses-*
*# sion is corrupted but key is not yet divulged or session is uncorrupted*
*# and query is for one of "our" partial keys.*

**if** $(i_{sid,grp} = \mathsf{undef})$ **then** *# New session*
**output:** $\mathsf{in}_{\mathsf{gdh}}!$ (getView, $|grp|$), $\mathsf{in}_{\mathsf{gdh}}{}^{\triangleleft}!$ (1);
**input:** $\mathsf{out}_{\mathsf{gdh}}?$ (getViewR, $i, view$);
$i_{sid,grp} \leftarrow i$; $view_{sid,grp} \leftarrow view$
**for all** $(v :: corr_v = \mathsf{true})$ **do** $ses_{v,sid,grp} \leftarrow \mathsf{corrupted}$; **end for**
**end if**
**if** $(\forall v \in grp : ses_{v,sid,grp} \neq \mathsf{corrupted})$ **then** *# Session uncorrupted*
$\beta' \leftarrow \mathsf{setbit}(\beta, \mathsf{idx}(grp, u)) :: (\beta, \gamma) \in view_{sid,grp}$;*# Index of exponentiation*
**output:** $\mathsf{out}_{\mathsf{gdhM},u}!$ (expR, $\gamma' :: (\beta', \gamma') \in view_{sid,grp}$), $\mathsf{out}_{\mathsf{gdhM},u}{}^{\triangleleft}!$ (1);
**else** *# Group contains a corrupted participant*
**if** $(secrets_{sid,grp} = \mathsf{undef})$ **then** *# Secrets not yet known*
**output:** $\mathsf{in}_{\mathsf{gdh}}!$ (getSecret, $i_{sid,grp}$), $\mathsf{in}_{\mathsf{gdh}}{}^{\triangleleft}!$ (1);
**input:** $\mathsf{out}_{\mathsf{gdh}}?$ (getSecretR, $secrets$);
$secrets_{sid,grp} \leftarrow secrets$;
**end if**
**output:** $\mathsf{out}_{\mathsf{gdhM},u}!$ (expR, $\gamma^{secrets_{sid,grp,\mathsf{idx}(grp,u)}}$); $\mathsf{out}_{\mathsf{gdhM},u}{}^{\triangleleft}!$ (1);
**end if**

**end transition**

**transition** $\mathsf{in}_{\mathsf{gdhM},u}?$ (getKey, $sid, grp, \gamma$)

**require:** $(u \in grp) \wedge (i_{sid,grp} \neq \mathsf{undef}) \wedge (ses_{u,sid,grp} \neq \mathsf{finished}) \ \wedge \ (((\exists \beta : (\beta,$
$\gamma) \in view_{sid,grp}) \wedge (\mathsf{setbit}(\beta, \mathsf{idx}(grp, u)) = 1^{|grp|})) \ \vee$
$((key_{sid,grp} = \mathsf{undef}) \wedge (\exists v \in grp : ses_{v,sid,grp} = \mathsf{corrupted})))$;
*# A legitimate caller of an initialized but unfinished session either asking*
*# for a correct key or being corrupted without somebody having asked*
*# for the ideal key before*

**if** $key_{sid,grp} \neq \mathsf{undef}$ **then** *# (Ideal) key already defined...*
*# ...so just return this key*
$ses_{u,sid,grp} \leftarrow \mathsf{finished}$;

**output:** $\mathsf{out}_{\mathsf{gdhM},u}!$ (getKeyR, $key_{sid,grp}$), $\mathsf{out}_{\mathsf{gdhM},u}{}^{\triangleleft}!$ (1);
**else** # *(Ideal) key does not yet exist and ...*
**if** ($\forall v \in grp : ses_{v,sid,grp} \neq \mathsf{corrupted}$) **then** # *... uncorrupted session*
**output:** $\mathsf{in}_{\mathsf{gdh}}!$ (getKey, $i_{sid,grp}$), $\mathsf{in}_{\mathsf{gdh}}{}^{\triangleleft}!$ (1);
**input:** $\mathsf{out}_{\mathsf{gdh}}?$ (getKeyR, $key$);
$key_{sid,grp} \leftarrow key$;
$ses_{u,sid,grp} \leftarrow \mathsf{finished}$; # *Mark only uncorrupted sessions as finished!*
**output:** $\mathsf{out}_{\mathsf{gdhM},u}!$ (getKeyR, $key_{sid,grp}$), $\mathsf{out}_{\mathsf{gdhM},u}{}^{\triangleleft}!$ (1);
**else** # *Group contains corrupted participants and (ideal) key undefined*
**if** ($secrets_{sid,grp} = \mathsf{undef}$) **then** # *Secrets not yet known*
**output:** $\mathsf{in}_{\mathsf{gdh}}!$ (getSecret, $i_{sid,grp}$), $\mathsf{in}_{\mathsf{gdh}}{}^{\triangleleft}!$ (1);
**input:** $\mathsf{out}_{\mathsf{gdh}}?$ (getSecretR, $secrets$);
$secrets_{sid,grp} \leftarrow secrets$;
**end if**
**output:** $\mathsf{out}_{\mathsf{gdhM},u}!$ (getKeyR, $\mathsf{h}(\gamma^{secrets_{sid,grp,\mathsf{idx}(grp,u)}})$), $\mathsf{out}_{\mathsf{gdhM},u}{}^{\triangleleft}!$ (1);
**end if**
**end if**
**end transition**

**transition** $\mathsf{in}_{\mathsf{gdhM},u}?$ (corrupt)
$corr_u \leftarrow \mathsf{true}$;
**for all** ($sid, grp :: (u \in grp) \wedge (i_{sid,grp} \neq \mathsf{undef}) \wedge (ses_{u,sid,grp} \neq \mathsf{finished})$)
**do**
$ses_{u,sid,grp} \leftarrow \mathsf{corrupted}$; # *Mark only locally unfinished sessions*
**end for**
**output:** $\mathsf{out}_{\mathsf{gdhM},u}!$ (corruptR), $\mathsf{out}_{\mathsf{gdhM},u}{}^{\triangleleft}!$ (1);
**end transition**

**transition** $\mathsf{in}_{\mathsf{gdhM},u}?$ (getSecret, $sid, grp$)
**require:** $(u \in grp) \wedge (i_{sid,grp} \neq \mathsf{undef}) \wedge (ses_{u,sid,grp} = \mathsf{corrupted}) \wedge (key_{sid,grp} = \mathsf{undef})$;
# *A legitimate caller of a started session and we are corrupted but the key*
# *has not been exposed*
**if** ($secrets_{sid,grp} = \mathsf{undef}$) **then** # *Secrets not yet known*
**output:** $\mathsf{in}_{\mathsf{gdh}}!$ (getSecret, $i_{sid,grp}$), $\mathsf{in}_{\mathsf{gdh}}{}^{\triangleleft}!$ (1);
**input:** $\mathsf{out}_{\mathsf{gdh}}?$ (getSecretR, $secrets$);
$secrets_{sid,grp} \leftarrow secrets$;

**end if**
**output:** $\mathsf{out}_{\mathsf{gdhM},u}!$ (getSecretR, $secrets_{sid,grp,\mathsf{idx}(grp,u)}$), $\mathsf{out}_{\mathsf{gdhM},u}{}^{\triangleleft}!$ (1);
**end transition** ◇

The following lemma shows that we safely replace the real system by the semi-real system.

**Lemma 2.5**

$$Sys^{\mathsf{gke,ika1}}_{n,tb,ct} \geq_{\mathsf{sec}} Sys^{\mathsf{gke,ika1,sr}}_{n,tb,ct}$$

□

*Proof.* Our goal is to show that the input-output behavior of the two systems is identical.

The biggest difference, of course, is the different number of machines in both systems. However, the existence of the sub-machines GDH_Mux and $\mathsf{GDH}^{(0)}_{n,mxkey}$ is hidden. The self-clocking and the use of secure connections for remote procedure calls in $Sys^{\mathsf{gke,ika1,sr}}_{n,tb,ct}$ ensures that the system control the scheduling for the whole duration of information flows through (honest) machines from H to A (and vice versa) and makes these flows externally visible as single atomic actions identical to $Sys^{\mathsf{gke,ika1}}_{n,tb,ct}$. This is also not violated by corruptions since the transparent mode does not leak any information on the existence of sub-machines.

Furthermore, it is easy to see that we mainly have to focus on the deterministic aspects. The only probabilistic actions of honest machines are the generation of the secret exponents, and they are chosen in both systems randomly as well as independently from the same distribution. The fact that the exponents are chosen in $Sys^{\mathsf{gke,ika1,sr}}_{n,tb,ct}$ by a submachine and also not at the same points in time as in $Sys^{\mathsf{gke,ika1}}_{n,tb,ct}$ does not matter. As argued above, the submachine is hidden. Additionally, the behavior of honest machines does not directly depend on these random choices. Due to this and the following argumentation on the deterministic behavior, externally visible events which are causally related to the generation of secret exponents are consistent with their corresponding events in $Sys^{\mathsf{gke,ika1}}_{n,tb,ct}$.

To see that the deterministic behavior in $Sys^{\mathsf{gke,ika1,sr}}_{n,tb,ct}$ is consistent with $Sys^{\mathsf{gke,ika1}}_{n,tb,ct}$, you should first observe that the external interface including **enabled if:** and **ignore if:** conditions is identical in both systems. The next

and most crucial step is to understand the **require:**-clauses in GDH_Mux. They ensure that, independent of the behavior of a calling $\mathsf{M}'_u$:

1. $\mathsf{GDH}^{(0)}_{n,mxkey}$ is consistently called, e.g., for all session at most one of getSecret and getKey is sent to $\mathsf{GDH}^{(0)}_{n,mxkey}$; and

2. all partial GDH keys and session keys returned to a caller of getKey and exp are consistent with the provided $\gamma$'s.[17]

However, these condition as well as the behavior of GDH_Mux should also not be too strict. They certainly have to ensure that:

3. calls by an uncorrupted $\mathsf{M}'_u$, in particular to getKey, do not block on a **require:** condition;

4. GDH_Mux provides an ideal key, i.e., one retrieved via getKey from $\mathsf{GDH}^{(0)}_{n,mxkey}$, for sessions where no group member is corrupted at the point of the first getKey;[18] and

5. "corrupted" keys, i.e., keys where the provided $\gamma$ does not match the expected value, should always be computed correctly using exponentiations to the given base $\gamma$.

If these conditions are fulfilled, clearly, an uncorrupted $\mathsf{M}'_u$ performs (in conjunction with GDH_Mux) the same state updates and behaves (as visible externally) identical to the corresponding $\mathsf{M}_u$. This holds also for corruptions since GDH_Mux provides the necessary information contained by $\mathsf{M}_u$ but lacking in $\mathsf{M}'_u$, i.e., exponents or keys which are not yet deleted.

This leaves us, finally, with the task of verifying that all of above conditions are fulfilled by GDH_Mux and $\mathsf{GDH}^{(0)}_{n,mxkey}$. Foremost, observe that $\mathsf{GDH\_Mux}.i_{sid,grp}$ uniquely relates sessions from $\mathsf{M}'_u$ (using the parameters $(sid, grp)$) with GDH instances provided by $\mathsf{GDH}^{(0)}_{n,mxkey}$ and identified by $i$. Furthermore, the tests $(u \in grp)$ ensure that only legitimate users of session are serviced. Let us address now the different conditions in turn.

[17] This does not necessarily mean that the session key must be identical to the key correctly derived from the GDH key corresponding to $\gamma$. It only requires that everybody providing the same $\gamma$ for a particular session will receive the same key. This is not important here but will be crucial when constructing the simulator.

[18] Again, this is not directly relevant here but crucial when constructing the simulator.

The validity of Condition 1 holds for the following reasons. Since honest machines always call $\mathsf{Gen}'$ before calling $\mathsf{GDH\_Mux}$, $\mathsf{GDH}^{(0)}_{n,mxkey}$ is appropriately initialized before $\mathsf{GDH\_Mux}$ calls it. Additionally, $\mathsf{GDH\_Mux}$ request GDH instances correctly on demand. Furthermore, a call to $\mathsf{getKey}$ is remembered in $key_{sid,grp}$. This caching as well as the similar caching of secret exponents ensures that $\mathsf{GDH}^{(0)}_{n,mxkey}$ is asked only once per session for either of them. Furthermore, the conditions ($key_{sid,grp} = \mathsf{undef}$) and the protocol flow guarantee that $\mathsf{getSecret}$ is never called after a call to $\mathsf{getKey}$. Similarly, $\mathsf{getSecret}$ is only called for corrupted sessions, a case in which $\mathsf{getKey}$ is never called (note that sessions cannot be "uncorrupted").

Condition 2 is trivially true since: (1) $\mathsf{GDH}^{(0)}_{n,mxkey}$ computes all keys based on real GDH partial keys and the correct key derivation, and (2) $\mathsf{GDH\_Mux}$ tests for "incorrect" $\gamma$'s, which cannot be found in the set of partial GDH keys, and computes the required value itself. (Note that this can only happen in case of a corruption and therefore calling $\mathsf{getSecret}$ is OK.)

Regarding Condition 3, note that honest machines $\mathsf{M}'_u$ pass always properly formated parameters. It is obvious that $\mathsf{GDH\_Mux}$ will not block on any **require:** condition for uncorrupted sessions. $\mathsf{exp}$ is called by each machine at least once before $\mathsf{getKey}$ is called a single time. Furthermore, the parameters are always correct and consistent with the GDH partial keys obtained by $\mathsf{GDH}^{(0)}_{n,mxkey}$ due to the honesty of machines and by construction of the protocol. This ensures that all exponentiations can be served from the GDH partial keys and, on calls to $\mathsf{getKey}$, the session is initialized but not terminated. Similarly, for sessions where some group members are corrupted beforehand, e.g., due to static corruptions, $\mathsf{GDH}^{(0)}_{n,mxkey}$ is never asked for $\mathsf{getKey}$. Therefore, $key_{sid,grp}$ remains undefined and exponentiations and key derivations do not block when the base $\gamma$ does not match the partial GDH keys. This covers the case of static corruption ($ct = \mathsf{static}$). To cover the case of adaptive corruptions ($ct = \mathsf{adaptive}$), it is sufficient to consider the following scenario: a uncorrupted group starts a session and later during the session a group member $\mathsf{M}'_u$ gets corrupted. First, note that dishonest machines never call $\mathsf{GDH\_Mux}$. Then, observe that, due to the confirmation flows, at the point of the first call to $\mathsf{getKey}$ no other member will call $\mathsf{exp}$ anymore for the same session. Let us now consider the two following two possible cases: In the first case, the session gets first corrupted *before* the first call to $\mathsf{getKey}$. In this case, $\mathsf{GDH\_Mux}$ marks the session as corrupted

and can safely retrieve the secret exponents $\mathsf{GDH}^{(0)}_{n,mxkey}$ by calling getSecret and serve (potentially inconsistent) queries to exp, getKey, and getSecret. (Note that getSecret or getKey might be called during corruption of $\mathsf{M}'_u$ or subsequent corruptions of other machines.) In the second case, the session gets corrupted only *after* the first call to getKey. Then, all exponents got previously deleted in all (then honest) machines $\mathsf{M}'_u$. Furthermore, neither exp nor getSecret will be called anymore. Since $key_{sid,grp}$ is cached and since, due to the confirmation flows, there exists an agreement among all machines on the $\gamma$ required as input to getKey, GDH_Mux can serve all subsequent getKey queries.

The fulfillment of Conditions 4 and 5 should be immediately clear from the **require:** condition for getKey messages. ∎

### 2.5.3 Replacing $\mathsf{GDH}^{(0)}_{n,mxkey}$ by $\mathsf{GDH}^{(1)}_{n,mxkey}$

In the next step, we replace $\mathsf{GDH}^{(0)}_{n,mxkey}$ by $\mathsf{GDH}^{(1)}_{n,mxkey}$. The rest of the system remains as in Figure 2.5. We call the resulting semi-ideal system $Sys^{\mathsf{gke,ika1,si}}_{n,tb,ct}$. The composition theorem from [2] and Lemma 2.4 immediately imply the following result:

**Lemma 2.6**

$$Sys^{\mathsf{gke,ika1,sr}}_{n,tb,ct} \geq_{\mathsf{sec}} Sys^{\mathsf{gke,ika1,si}}_{n,tb,ct}$$

□

### 2.5.4 Security of the System with $\mathsf{GDH}^{(1)}_{n,mxkey}$ with Respect to the Ideal System

We now define as a final step the simulator as a variant of the previous system.

**Scheme 2.5 (Simulator for Scheme 2.2)**

The overall structure of the simulator $\mathsf{Sim}_{\mathcal{H}}$ is shown in Figure 2.6.

The submachine Gen′ of $\mathsf{Sim}_{\mathcal{H}}$ is identical to its counterpart in the semi-real and semi-ideal systems. Each submachine $\mathsf{M}''_u$ of $\mathsf{Sim}_{\mathcal{H}}$ has the same ports as its semi-real counterpart $\mathsf{M}'_u$, except that its ports are connected to

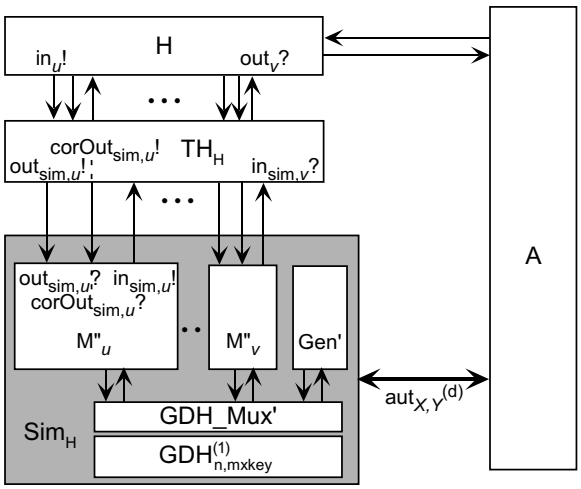


Figure 2.6: Simulator

$\mathsf{TH}_{\mathcal{H}}$ and correspondingly renamed, i.e., $\mathsf{in}_u?$ becomes $\mathsf{out}_{\mathsf{sim},u}?$, $\mathsf{out}_u!$ becomes $\mathsf{in}_{\mathsf{sim},u}!$, and $\mathsf{corrupt}_u?$ becomes $\mathsf{corOut}_u?$ for all $u \in \mathcal{M}$. Furthermore, the domain of the variable $key_{sid,grp}$ is extended to the value ideal, a value which is distinct from $\{0,1\}^k$, $G$ and undef and has an empty transport encoding. $\mathsf{M}''_u$ also has the same state-transition function as $\mathsf{M}'_u$ except for this renaming and the following changes:

- the message type key is everywhere replaced by finish. Note that a message of type finish with a parameter ideal as third parameter will result, due to above mentioned encoding properties, in a two-parameter message only (allowing the TH to choose an ideal key).
- $\mathsf{M}''_u$ expects a message (state, *state*) instead of (do) on port $\mathsf{corrupt''}_u?$. This corruption message is also passed to $\mathsf{in}_{\mathsf{gdhM},u}!$.

Submachine GDH_Mux′ is identical to GDH_Mux except

- The domain of the variable $key_{sid,grp}$ is extended to the value ideal.
- Instead of calling getKey to $\mathsf{GDH}^{(1)}_{n,mxkey}$ in transition getKey it defines $key_{sid,grp}$ always as ideal: This will result in a finish message with no key and allow $\mathsf{TH}_{\mathcal{H}}$ to choose the key as desired in the absence of corrupted parties. (Note that due to the program logic no call to getKey from

encode_state() during a corruption will ever return ideal, so no adversary will be confused by an unexpected value ideal.)

- It expects the state *state* of the corrupted party as a parameter of message corrupt, extracts all session keys from *state* and assigns them to the corresponding variable $\mathsf{GDH\_Mux}'.key_{sid,grp}$.

$\diamond$

As the following lemma shows, the semi-ideal system is at least as secure as the ideal system.

**Lemma 2.7**

$$Sys^{\mathsf{gke,ika1,si}}_{n,tb,ct} \geq_{\mathsf{sec}} Sys^{\mathsf{gke,ideal}}_{n,tb,ct}$$

$\square$

*Proof.* The proof of this lemma is quite similar to the proof of Lemma 2.5. The difference in the structure among the two systems is hidden for the same reasons given in that lemma. Similar arguments hold regarding the probabilistic aspects, except that now the ideal key is generated by $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{GDH}^{(1)}_{n,mxkey}$, respectively. This leaves the deterministic aspects.

The same argumentation found in Lemma 2.5 ensures also that the sub-machines $\mathsf{M}''_u$ of $\mathsf{Sim}_{\mathcal{H}}$ interoperate consistently with $\mathsf{GDH\_Mux}'$, and $\mathsf{GDH}^{(1)}_{n,mxkey}$. The main question to answer is whether the interposition of $\mathsf{TH}_{\mathcal{H}}$ does not result in observable differences of behavior. It is easy to verify, that the messages exchanged on the connections between $\mathsf{TH}_{\mathcal{H}}$ and uncorrupted sub-machines in $\mathsf{Sim}_{\mathcal{H}}$ match the required message format. For corrupted sub-machines, the logic of the specification ensures that the corresponding "virtual sub-machine" in $\mathsf{TH}_{\mathcal{H}}$ is switched to transparent mode at the same time, too. Furthermore, it should be clear that $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{Sim}_{\mathcal{H}}$ keep session-specific state and the corruption status of users in lock-step. This means for most cases, $\mathsf{TH}_{\mathcal{H}}$ will safely route messages forth and back between $\mathsf{M}''_u$ and the corresponding user $u$. The only real question is whether a finish is accepted by $\mathsf{TH}_{\mathcal{H}}$ and results in appropriate assignment of session keys. However, this is ensured mainly due to the fulfillment of the Conditions 2, 4 and 5 mentioned and shown to hold in the proof of Lemma 2.5. For sessions which are already corrupted *before* the first getKey occurs, the "real" session key derived from the GDH keys are passed by $\mathsf{GDH\_Mux}'$ to

$\mathsf{M}''_u$. Furthermore, as the session is corrupted $\mathsf{TH}_{\mathcal{H}}$ will accept this key in a finish message. For sessions which get corrupted only *after* the first call to getKey, we are forced to finish the session with an ideal key. This works for following reasons: (1) no traces of the "real" session key exist anymore, (2) the corresponding session key returned by $\mathsf{GDH\_Mux}'$ to $\mathsf{M}''_u$ on a call to getKey will have the value ideal, and (3) the sending of a finish message with key ideal results, as noted in the description of the simulator, in a finish message with no third parameter as required by $\mathsf{TH}_{\mathcal{H}}$ to accept that session and to generate the concrete ideal key itself.

■

*Proof.* (of Theorem 2.1) The result follows immediately from Lemmas 2.5, 2.6 and 2.7, and the fact that "$\geq_{\mathsf{sec}}$" is transitive [2]. Applying Remark 2.1 to Theorem 2.4 (the number *numpkey* of different partial keys visible to an adversary in Scheme 2.2 is $(n(n-1)/2)-1$) and observing that only Lemma 2.6 involves computational security, we achieve the following overall concrete security: Given a distinguisher which breaks Scheme 2.2 in time $t$ and with success probability $\epsilon$, we can break $(1-1/\mathsf{poly}(k))$-DDH(c:u; g:m; f:nsprim,o) with a time complexity of at most $(t + O(mxkey\, n^3\, k^3))(O(n^2 k)/\epsilon^2)$ and with overwhelming success probability. ■

## *2.6 Conclusion*

In this chapter we presented a formal model for group key agreement and corresponding protocols, which are important building blocks of many MAFTIA applications. The formal model is a rigorous specification of the group key agreement service (ideal system) and the protocols are rigorously defined by means of protocol machines (real system). Furthermore, we gave a rigorous pen-and-paper proof that this real system is as secure as the ideal system, i.e., it our protocol machines implement the specified secure group key agreement service.

As already the specification of the desired service (ideal system) is quite complex and, therefore, potentially prone to errors, we have chosen to model it in CSP and model-check it in order to increase the confidence in the correctness of our service specification. The modelling and model-checking of this specification is presented in the following chapter. It additionally in-

creases the level of confidence in the security of these protocols and it is a further example for the combined use of cryptographic proofs and automated formal methods.

# 3 CSP Model of GKE

In this chapter we describe our CSP modeling of the Ideal System for Group Key Establishment (GKE) of chapter 2.

## *3.1 Background*

One important aim of MAFTIA Work package 6 is to bridge the gap between the formal *rigorous secure reactive systems* theory of Pfitzmann and Waidner [4] and the world of automated, or 'tool-assisted' verification.

The rigorous secure reactive systems theory of [4] is a well-defined mathematical framework for faithfully modeling cryptographic systems. In that theory, both 'real world' and 'idealized' cryptographic systems are transcribed as *structures*, $(M, S)$, where $M$ is a set of probabilistic state-transition machines, and $S$ is a set of *specified ports* - essentially the user-interface channels. A structure, $(M, S)$, may be augmented to a $configuration$, $(M, S, H, A)$, by specifying a set of users, $H$, and an adversary machine, $A$. A configuration is 'runnable' in the sense that it yields a probability space of traces in terms of the communications over its $ports$ (channels), and we may refer to the 'view of a system' with respect to the probability space of traces over certain *specified* ports.

A *Trusted Host* is a structure, in the above sense, representing the 'best possible' service - a specification of a service suffering tolerable (and possibly unavoidable) imperfections. In the 'real world' however, any particular implementation of that service will invariably suffer from more malignant imperfections - because it will be reliant, for example, on imperfect cryptographic primitives, or 'lossy' communications mediums.

The question then arises as to how to measure the security of these 'real world' systems against the 'best possible' service, when the former, if reliant on imperfect primitives, can never quite match the Trusted Host. In the world of automatic, tool assisted verification that is usually a non-question, because the type of 'imperfections' that we are here referring to are usually abstracted away by Dolev-Yao type assumptions. In [4], however, the semantics of the cryptographic primitives is faithfully preserved, and there is a formal mathematical notion of *simulatability* – a precise definition of what it means for a 'real world' system to be 'as least as secure as' the 'best possible' service

as specified by the Trusted Host. That is, if $Sys$ is real system and $TH$ is a Trusted Host specification, then $Sys$ is as least as secure as $TH$, written $TH \leq_{sec} Sys$, if for any configuration of $Sys$ with users $H$ and adversary $A$, there is a configuration of $TH$ with users $H$ and adversary $A'$, such that the two configurations are indistinguishable at the user interface[1].

Clearly the notion of simulatability has parallels with the notion of refinement, or, more specifically, bisimilarity, in the CSP calculus ($Spec \sqsubseteq Impl$). It was natural, then, for WP6 to ask whether it might be possible to use automated checkers, such as FDR, to prove simultability relationships. That would be good, because proving simultability relationships by hand requires a great deal of human effort (which is inevitably open to error).

However, that goal is very ambitious. The problem is not specifically one of theory - there is no reason why a Trusted Host or $Sys$ configuration could not naïvely be transcribed in CSP or similar calculi.[2] Rather, the problem is more one of practicality, in that the state-space of the resulting models would invariably be intractably large for the automatic checkers (as evinced by our discussion in 3.2, below), and it is our opinion that that will remain the case for the foreseeable future.

The problem, then, has to be addressed by advancing the theory of state-space optimizations[3] and their practical application to the particular type of probabilistic state-transition machines that we see in [4]. We believe that this is achievable, but that the effort and time scales involved go beyond this project. Nevertheless, this chapter was written as a kind of 'feasibility study' into the viability of using CSP and FDR to specify the type of state machines we see in the rigorous secure reactive systems theory of [4], and to verify meaningful properties of those machines that would otherwise have to be done by hand. As such, this work may be considered complementary to that of Backes, Jacobi and Pfitzmann [7], in which the PVS theorem prover was used to verify certain non-cryptographic, 'book-keeping' mechanisms of a scheme for ordered secure message transmission.

[1] I.e. that the probability spaces of traces restricted to the user ports are bisimilar.

[2] Probably the biggest issue would be accounting for the probabilities associated with transitions in the state machines. However, there are available probabilistic model checkers such as Prism [47], and one may even model probabilistic behavior in finite-state $CSP_M$ (although at a very high cost in terms of state-space).

[3] E.g. compression operators, watchdog transformations, partial order reductions, symmetry breaking, and D.I. arguments.

For our 'feasibility study' we chose, as an example, to model in CSP the Trusted Host for Group Key Establishment of chapter 2. Foremost, this is an exercise in transcribing a complex state machine - the Trusted Host - into CSP that is optimised with respect to state-space, completing a compilable model of the Trusted Host was our priority. However, we would like also to use that model to verify some 'informal properties' asserted of Trusted Host using FDR – in chapter 2, those properties are proven by hand. The Trusted Host is, of course, a specification for Group Key Establishment; in chapter 2 it is used to mathematically prove the security properties of one of the *Initial Key Agreement* protocols of the *CLIQUES* protocol suite. Although these 'informal properties' may not be transparently true of the Trusted Host, their verification alone would hardly have justified the effort of transcribing the Trusted Host in CSP. The transcription of the properties then - discussed in 3.11 - is of secondary concern to us.

## *3.2 Overview of Model*

The GKE Ideal System is parameterized by:

- the number of participants, $n$
- the a-priori uncorrupted participants, $H$
- the bound on key length, $k$
- the set of session identifiers, $SID$
- the corruption model, $ct$ (which is *static* or *adaptive*)

We now discuss the main issue that arose in our modeling of the GKE Ideal System, i.e. state-space explosion.

It was a simple matter to derive the following formulae for the number of variables of the Ideal System, and an *upper bound* on the number of configurations that those variables can be in, in terms of $n$, $k$ and the cardinality of $SID$, $|SID|$:

$\#variables = n^2 + 3 \times 2^n \times |SID| \times n$

$total\,\#configurations = 4^{n^2} \times 3^{2^n \times |SID| \times n} \times (2^k + 1)^{2^n \times |SID| \times n} \times (2^n \times |SID|)^{2^n \times |SID| \times n}$

We used the above formulae to calculate $\#variables$ and $\#configurations$ for $1 \leq n \leq 3$ and $1 \leq k, |SID \leq 2$. The results are tabulated in 3.1, below, where a '?' in a column means the value was too large for the computer to calculate.

| $n$ | $k$ | $\|SID\|$ | $\#variables$ | $\#configurations$ |
|---|---|---|---|---|
| 1 | 1 | 1 | 7 | 1296 |
| 1 | 2 | 1 | 7 | 3600 |
| 1 | 1 | 2 | 13 | 6718464 |
| 1 | 2 | 2 | 13 | 51840000 |
| 2 | 1 | 1 | 28 | 7.222041e14 |
| 2 | 2 | 1 | 28 | 4.299817e16 |
| 2 | 1 | 2 | 52 | 1.335242e32 |
| 2 | 2 | 2 | 52 | 4.733037e35 |
| 3 | 1 | 1 | 81 | ? |
| 3 | 2 | 1 | 81 | ? |
| 3 | 1 | 2 | 153 | ? |
| 3 | 2 | 2 | 153 | ? |

Table 3.1: $\#variables$ and $\#configurations$ for small $n$, $k$ and $|SID|$

At first sight, the values tabulated in 3.1 suggest that it will only be feasible to model the GKE Ideal System tract-ably for very small, naïve values of $n$, $k$ and $|SID|$. However, that is probably overly pessimistic, bearing in mind that many of the variable states (configurations) may not actually be reachable in practice. This is a well known modeling phenomenon. We account for it, as best we can, as follows.

For every variable $X$ in the Ideal System, we have a single process, $VARIABLE\_X(X, init\,value)$, that is responsible for recording the current state of variable $X$. Other (non-$VARIABLE\_X$) processes may read $X$ and write to $X$ by synchronizing on special $get\_.X?\_$ and $set\_.X!\_$ events[4].

A single process, $VARIABLES$, is composed from the $VARIABLE\_X(X, init\,value)$ processes by combining those processes in various shared parallel combinations. The parallel combinations depend

[4]The $set\_$ channel name has a trailing underscore to avoid clashes with the FDR keyword $set$; the $get\_$ channel name has a trailing underscore for consistency.

on the type of the variables, and are such that the processes share as few events as possible (ideally, they would share no events and so be interleaved), c.f. 3.8 figure 3.6.

The $VARIABLES$ process is then put in shared parallel with a (stateless) Trusted Host process, $TH\_H$, and a 'random' key generation process, $KEYR$. The processes synchronize on the $get\_$, $set\_$ and a few other events, then the $get\_$ and $set\_$ events are hidden:

$$\begin{array}{l}( \; ( \; TH\_H \; [|\ldots|] \; KEYR \; ) \\ \qquad [|\{|\ldots, get\_, set\_|\}|] \\ \quad VARIABLES \; ) \\ \setminus \{|\ldots, get\_, set\_|\}\end{array}$$

The net effect of this construction[5] is to move the bulk of the work from FDR's compilation stage to the more efficient state-exploration phase, the latter phase is *not* impacted upon by non-reachable states.

Moreover, by breaking the system up like this, we are free to optimize the $VARIABLES$ process (which has no direct parallel in chapter 2) without obfuscating the $TH\_H$ process - that is good, as we want the latter to be easily traceable to the definition of the Trusted Host (as a state-transition machine) in chapter 2.

In the case of the GKE Ideal System, however, there are some additional complications due to the fact that there is existentional and universal quantification over the state variables. We defer discussion of this until 3.8.

## *3.3 Datatypes*

The GKE Ideal System assumes $n, tb \in \mathbb{N}$ and $ct \in \{static, adaptive\}$ are given. Here $n$ is the number of participants, $tb$ is a bound on the number of 'activations per port' - in effect, a bound on I/O buffer size - and $ct$ is the type of corruptions to be tolerated. In addition, there is a given $k \in \mathbb{N}$, the bound on key length, and a given set of session identifiers, $SID$.

Then $M$, the set of participant identifiers is defined to be $\{1..n\}$, and the set of keys is defined to be $\{0,1\}^k$. For brevity, we also defined $GRP$ to be $\mathbb{P}(M)$. In the $\mathrm{CSP}_M$, we have:

[5]It is regarded as a standard CSP construction.

```
M = {1..n}
MxM = {u.v|u<-M,v<-M}
-- User machines


GRP = Set(M)
-- The possible groups


Keys(0) = {<>}
Keys(n) = { x^<0>, x^<1> | x <- Keys(n-1) }
-- The set of session keys of length n.
```

The GKE Ideal System is a family of structures (in the sense of [4]) of single *Trusted Host* state-transition machines, $TH_H$, parametrized by $H \subseteq M$. Each $H$ represents a set of 'a-priori uncorrupted' participants. For each $H \subseteq M$, there is a corresponding $A = M - H$, representing the a-priori corrupted participants. In the $\mathrm{CSP}_M$, we have, for example:

```
nametype H = {1..n-1}
-- a-priori uncorrupted participants


A = diff(M,H)
```

and $TH_H$ is represented by the process TH_H.

Our model of the GKE Ideal System is parameterized by $n$, $k$, $ct$, $H$ and $SID$, although we expect the model to be tractable to FDR for only small instances of $n$, $k$ and $SID$; $tb$, if it appears at all, will be fixed at 1. The model was not written so as to be data independent in any of the parameters. Although it is quite plausible that the model could be written so as to be data independent, the effort involved was not considered worthwhile - given that the threshold values on the parameter data types would likely be so big as to result in intractable models.

## *3.4 Trusted Host state variables*

The Trusted Host machine has four types of variables:

- *state* variables $state_{u,v}$, $(u, v) \in M.M$, ranging over $\{undef, wait, init, corrupted\}$.

- *ses* variables $ses_{u,sid,grp}$, $(u, sid, grp) \in M.SID.GRP$, ranging over $\{undef, init, finished\}$.
- *key* variables $key_{u,sid,grp}$, $(u, sid, grp) \in M.SID.GRP$, ranging over $\{0,1\}^k \cup \{undef\}$.
- *prev* variables $prev_{u,sid,grp}$, $(u, sid, grp) \in M.SID.GRP$, ranging over $SID.GRP$.

In the CSP$_M$, the following general datatype was defined:

```
datatype keywords =
   ses |
   finish.SID.GRP.union(Keys(k),{no_key(k)}) |
   static |
   adaptive |
   undef |
   wait |
   corrupted |
   finished |
   init |
   initialized.M |
   do |
   key.SID.GRP.union(Keys(k),{undef_key}) |
   new.SID.GRP.union({not_present},{(sid',grp')|sid'<-SID,grp'<-GRP}) |
   state
```

A few concessions were necessary in order to keep the scripts type correct. In particular, we had to define constants (of compatible types) to represent optional parameters (*not_present*) and undefined keys (*undef_key*). Also, we needed to append trailing underscores to the variable type tags.[6] The CSP$_M$ TH_H variable types were then defined as follows:

```
datatype variable_types =
   state_.M.M.{undef,wait,init,corrupted} |
   ses_.M.union(SID,{ -1}).GRP.{undef,init,finished} |
   key_.M.SID.GRP.union(Keys(k),{undef_key}) |
   prev_.M.SID.GRP.{ (sid',grp') | sid'<-SID, grp'<-GRP }
```

[6]This was necessary in order to ensure that the datatypes 'keywords' and 'variable_types' were both rectangular. We could not do this *and* have single tags for *state*, *ses* and *key*.

## 3.5 Functions

There were two GKE-specific functions.

The predicate $present(x)$ (and $present\_(x)$, for type correctness) returned true if $x$ was 'present', i.e. not one of the constants representing a 'not present' optional parameter. Otherwise it returned false.

```
present(x) = if x==not_present then
   false
else
   true

present_(key_u_sim) = if key_u_sim==no_key(k) then
   false
else
   true
-- Is an optional parameter present?
```

The function $int2bin : N \rightarrow Keys(k)$ was used for 'key generation', $int2bin(n)$ being the 'unpadded' binary representation of n as a $\text{CSP}_M$ sequence of 0s and 1s of length $k$.

```
Pad(0) = <>
Pad(n) = Pad(n-1)^<0>
int2bin(0) = <0>
int2bin(1) = <1>
int2bin(n) =
if n%2==0 then
   int2bin(n/2)^<0>
 else
   int2bin(n/2)^<1>
-- 'Key generation'
```

## 3.6 Channels

The GKE Trusted Host has three types of ports to users $u \in H$, these are $in_u?$, $out_u!$ and $corrupt_u?$. There are also three types of ports to the adversary, these are $in_{sim,u}?$, $out_{sim,u}!$ and $corOut_{sim,u}?$

For $u \in H$, inputs over port $in_u?$ are of the form $in_u.m$ where $m$ is '*init*' or '*new*'. The meaning of $in_u.init$ is that user $u$ is initializing; the meaning of $in_u.new$ is that $u$ is initializing a new session, or extending a previous one if optional parameters are present.

For $u \in H$, outputs over port $out_u!$ are of the form $out_u.m$ where $m$ is either $initialized.v$ for some $v \in M$, $key.sid.grp.x$ for $sid.grp.k \in SID \times GRP \times Keys(k)$, or $m$ is arbitrary. The meaning of $out_u.initialized.v$, $x \in M$, is that user $v$ is initialized from $u$'s point of view. The meaning of $out_u.key.sid.grp.x$ is that new key $x$ has been agreed for the session $sid$. The 'arbitrary outputs' over $out_u!$ are 'leaks' by a corrupted $u$, these outputs could only occur in the adaptive corruption model.

For $u \in H$, the only input over port $corrupt_u?$ is $corrupt_u.do$, meaning that user $u$ is now corrupted.

The channels $in_u?$, $out_u!$ and $corrupt_u?$ are defined in $\text{CSP}_M$ as follows:

```
channel in : M.{|init,new|}

channel out : M.{|initialized,finish,key|}

channel corrupt : M.{do}
```

For $u \in H$, inputs over port $in_{sim,u}?$ are of the form $in_{sim,u}.m$ where $m \in \{|initialized, finish|\}$ or $m$ is arbitrary. The meaning of $in_{sim,u}.initialized.v$, $v \in M$, is that user $u$ should consider $v$ as initialized. The meaning of $in_{sim,u}.finish.sid.grp.[key_{u,sim}]$, where the $key_{u,sim}$ parameter is optional, is that the adversary completes the session for $u$, assigning $key_{u,sim}$ to $u$ if that parameter is present. The 'arbitrary inputs' are from the adversary to a corrupted $u$, these can only occur in the adaptive corruption model.

For $u \in H$, outputs over port $out_{sim,u}!$ are of the form $out_{sim,u}.m$ where $m \in \{|init, new|\}$ or $m$ is arbitrary. The meaning of $out_{sim,u}.init$ is that user $u$ is initializing. The meaning of $out_{sim,u}.new.sid.grp.[sid'.grp']$, where the parameter $sid'.grp'$ is optional, is that user $u$ has initialized a new session. The arbitrary outputs over $out_{sim,u}!$ are arbitrary message-sends by corrupted $u$, these can only occur in the adaptive corruption model.

For $u \in H$, the only output over port $corOut_{sim,u}?$ is $corOut_{sim,u}.m$, where $m$ is, basically, an encoding of the current values of the TH variables corresponding to $u$. This channel was simplied in our model so that it just conveyed tuples $(sid, grp, key_{u,sid,grp})$, the rationale for this given in 3.9.

The channels $in_{sim,u}?$, $out_{sim,u}!$ and $corOut_{sim,u}?$ are defined in $\text{CSP}_M$ as follows:

```
channel corOut_sim : M.SID.GRP.Keys(k)

channel in_sim : M.{|initialized,finish|}
```

```
channel out_sim : M.{|init,new,initialized,finish|}
```

There are a number of $\text{CSP}_M$ channels additional to the above that have no corresponding ports in chapter 2, these channels are as follows.

The *keyR* channel that is used to return 'random' keys. This is discussed in more detail in 3.7. The *keyR* channel is defined in *TH.csp* as follows:

```
channel keyR : Keys(k)
-- Random keys
```

The *get_*, *set_* and $SET\_key\_$ channels; the $begin\_exists\_x$, $end\_exists\_x$ and $satisfied\_x$ channels, $x$ = 1,2 and 3; the $begin\_forall\_1$, $end\_forall\_1$ and $not\_satisfied$ channels. These channels are discussed in more detail in 3.8. They are defined in the *variables.csp* file as follows:

```
channel get_ : variable_types
-- Get the current value of a variable

channel set_ : variable_types
-- Set the value of a variable

channel SET_key_ : SID.GRP.Keys(k)
-- Assign the value of a key to all members of a group.


...


-- Does there exist a v in grp such that state_u,v != init?

channel begin_exists_1 : GRP.M
channel end_exists_1 : Bool
channel satisfied_1

...


-- Does there exist a v in grp such that state_v,v == corrupted or v in A?

channel begin_exists_2 : GRP.GRP
channel end_exists_2 : Bool
channel satisfied_2

...
```

```
-- Do there exist v0,v1 in grp such that ses_v0,sid,grp!=undef and
-- ses_v1,sid,grp!=undef and prev_v0,sid,grp!=prev_v1,sid,grp?

channel begin_exists_3 : SID.GRP
channel end_exists_3 : Bool
channel satisfied_3 : M.SID.GRP.Bool.{ (sid',grp') | sid'<-SID, grp'<-GRP }

...


-- Is it true that for all v in grp, ses_v,sid,grp!=finished?

channel begin_forall_1 : SID.GRP
channel end_forall_1 : Bool
channel not_satisfied
```

## 3.7 $KEYR$

The $KEYR$ process is a simple process that returns new 'random' keys over the channel $keyR$, below.

```
channel keyR : Keys(k)
-- Random keys
```

There are two versions of the $KEYR$ process, the first of which is as in Figure 3.1 below. This $KEYR$ keeps a counter, $n$, which is initialized to 0 and may be incremented up to a maximum value of $2^k$. At any time $n < 2^k$, $KEYR$ offers its environment $keyR.x$, where $x$ is the binary representation of $n$ as a $\text{CSP}_M$ sequence of 0s and 1s, 'padded' to length $k$. If and when the $keyR.x$ event is engaged in, $KEYR(n)$ recurses to $KEYR(n+1)$. With reference to Figure 3.1, '$\#x$' is the length of the sequence $x$, $Pad(k-\#x)$ is a sequence of $k-\#x$ 0s, and '$^\wedge$' is the $\text{CSP}_M$ concatenation operator.

$$
\begin{array}{l}
KEYR(n) = \\
\mathit{let} \\
\qquad x \;=\; \mathit{int2bin}(n) \\
\mathit{within} \\
\qquad \#x \leq k \;\&\; (\; keyR!Pad(k-\#x)^\wedge x \;\rightarrow\; KEYR(n+1)\;)
\end{array}
$$

Figure 3.1: First version of 'random' key generation process, $KEYR$

The second version of $KEYR$ is as in Figure 3.2, below. This $KEYR$ keeps a record of 'used' keys by way of parameter $used$; $used$ is initially

empty, and will always be some subset of $Keys(n)$. At any time that there are 'unused' keys remaining, i.e. $used \neq Keys(n)$, $KEYR$ offers its environment some $keyR.x$, where $x \in Keys(n) \setminus used$. Immediately subsequent to a $keyR.x$ event being engaged in, $KEYR(used)$ recurses to $KEYR(used \cup \{x\})$.

$$KEYR(used) = \\ keyR?x : diff(Keys(n), used) \rightarrow KEYR(union(used, \{x\}))$$

Figure 3.2: Second version of 'random' key generation process, $KEYR$

In 3.11 we aim to verify that our model of the Ideal System exhibits the security property of 'Key Secrecy'. That property requires session keys to be generated freshly and randomly. We now briefly discuss the issue of whether either of the above models are refined enough to capture that requirement.

The first version of $KEYR$ 'generates' up to $2^k$ keys in order: $x_0, x_1, x_2, \ldots, x_{2^k-1}$, where $x_n$ is the binary representation of $n$ as a $\text{CSP}_M$ sequence of 0s and 1s. This version is the more economical in terms of state space - it requires only $2^k$ states, whereas the second version requires $2^{2^k}$.[7] However, if we use this version of $KEYR$, then it is conceivable that a genuine design fault is hidden from us, in that in some behaviors of the Ideal System it may be the case that the order in which the keys were generated was predictable solely because of some idiosyncrasy of the Trusted Host process, but this happend also to be the (fixed) order in which the $KEYR$ process offered the keys to its environment. That is something of a mute point though, because we can easily see from the transcription of the $TH\_H$ transitions, detailed in 3.9, that the only reference $TH\_H$ makes to the channel $keyR$ is the unconstrained input '$keyR?sk$'. Thus, the $TH\_H$ process could not possibly influence the order in which keys were 'generated' - it merely takes whatever happens to be offered by $KEYR$.

However, fixing the order of a supposedly 'random' key sequence is counter intuitive. It may be possible to use 'off-line' arguments, as above, to assert that the resulting model of the Ideal System still satisfies the criteria

[7] This difference is significant - even for very small $k$ - when one bears in mind that the size of $KEYR$'s state space is a multiplicative factor in the overall state space size of our model.

that keys are, for all intents and purposes, 'random', but the model is no longer the generic specification that it is intended to be. The second version of $KEYR$ is, therefore, a more natural model in that it affects fresh and 'random' key generation, at least insofar as there being no preferred 'next key' (among the unused keys). Moreover, as we shall see in 3.11, if the order is not fixed, then our completed model of the Ideal System is symmetric in the keys, and that allows us to make some short cuts when specifying and verifying certain properties of the system.

The question remains, though, as to whether an environmental choice of 'next key' (with reference to Figure 3.2) is sufficient, or whether that choice has necessarily to be non-deterministic. This is more subtle. The difference manifests itself in the willingness of our model to refuse to offer some keys as 'next key'. With a non deterministic choice, it may refuse some keys. However, there is no informal requirement for that property. Also, the Ideal System is intended as a stand-alone specification – it is not used in the context of some other process' environment. This means that the only processes that refer to the channel $keyR$ are $KEYR$ and $TH\_H$, so the 'next key' is, in effect, a free choice from the unused keys. We therefore contend that the second version of $KEYR$ (that of Figure 3.2) is an adequate model.

In chapter 2, random key generation is implicit and there is no machine corresponding to $KEYR$.

## *3.8* $VARIABLES$

The $VARIABLES$ process has no corresponding machine in chapter 2. The process should be regarded as pure $\text{CSP}_M$ support for the referencing of TH_H variables that is implicit in the definition of TH_H as a state-transition machine in chapter 2.

The Trusted Host state-transition machine of chapter 2 implicitly refers to its variables and performs existential and universal quantification over the variables as part of conditional tests; it also updates variables in the process of transitioning. This is done explicitly in the $\text{CSP}_M$ by the $VARIABLES$ process.

When writing $VARIABLES$, we were particularly concerned with making the process as efficient as possible with regards to state-space. As discussed in 3.2, optimising the state-space of $VARIABLES$ was crucial to the

tractability of the model of the Ideal System as a whole, this is reflected in the fact that actually more effort was spent on developing the $VARIABLES$ process than the TH_H process itself.

The $VARIABLES$ process is composed of copies of a generic $VARIABLE\_X(X, init\,value)$, one copy for each variable $X$ of TH_H, in various shared parallel combinations.

The parameter $x$ of $VARIABLE\_X(X, x)$ records the current value of $X$. The process consists of a number of environmental choices. Two of these choices are unguarded. The remaining choices are guarded according to the type of the parameter $X$. For example, we have the guard $\Box\{|X|\} \leq \{|state\_|\}$ & which ensures that that choice will be compiled if and only if the parameter $X$ is a variable of type $state\_$ ($=state$).

The two unguarded environmental choices are those that perform the basic reading and writing to $X$ over the $get\_$ and $set\_$ channels as in 3.3.

$(\, get\_!X!x \rightarrow VARIABLE\_X(X, x)\,)$
$\Box$
$(\, set\_!X?y \rightarrow VARIABLE\_X(X, y)\,)$

Figure 3.3: Reading and writing to $X$

In addition to simple conditional tests performed on individual variables, some of the TH transitions are dependent on existential and universal quantification over sets of variables. However, quantifying over variable sets is not a straightforward matter given that the shared parallel copies of $VARIABLE\_X(X, x)$ do not, of course, share process parameters. To model the $\exists$s and $\forall$s we have special 'control' processes that basically 'query' each individual $VARIABLE\_X(X, x)$ process, which, in turn, 'report back' as to whether or not $X$ currently satisfies the $\exists$ or $\forall$ predicate. There is one guarded choice for each $\exists$ and $\forall$.

To illustrate how we achieve $\exists$ quantification, let us consider the predicate '$\exists v \in grp : state_{u,v} \neq init$' that occurs in chapter 2.

In the $VARIABLE\_X(X, x)$ process the guarded choice relevant to the above $\exists$ is as in 3.4. The choice will be compiled if and only if $X$ is a variable of $state$ type. Suppose that that is the case, and that $X \equiv state\_u'.v$ for some $u', v \in M$.

$\Box\,\{|X|\} \leq \{|state\_|\}$ & (
    $begin\_exists\_1?grp : GRP?u : M \rightarrow$
    $if\,(U(X) == u\ and\ member(V(X), grp)\ and\ x \neq init)\ then$ (
        $satisfied\_1 \rightarrow$
        $end\_exists\_1?\_ \rightarrow$
        $VARIABLE\_X(X, x)$
    )
    $else$ (
        $end\_exists\_1?\_ \rightarrow$
        $VARIABLE_X(X, x)$
    )
)

Figure 3.4: Does $X \equiv state_{u.v}$ satisfy the predicate $v \in grp\,and\,state_{u.v} \neq init$

With reference to 3.4, suppose that the 'trigger' event $begin\_exists\_1?grp : GRP?u : M$ has occurred. Then, given that $U(X)$ returns $u'$ and $V(X)$ returns $v$, we see that the $if$ conditional evaluates to true if and only if $X \equiv state\_.u.v$, $v \in grp$, and the current value of $X$, i.e. $x$, is not equal to $init$. This is precisely the predicate '$v \in grp\,and\,state_{u,v} \neq init$'.

If the conditional evaluates to true, then $VARIABLE\_X(X, x)$ is prepared to engage only in '$satisfied\_1 \rightarrow end\_exists\_1?\_$' before recursing. If the conditional evaluates to false, then it is prepared to engage only in '$end\_exists\_1?\_$' before recursing.

The $VARIABLE\_X(X, x)$ processes, $X$ of type $state$, are then put in parallel so that they share as few $begin\_exists\_1$ and $end\_exists\_1$ events as possible (they do not share $satisfied\_1$ events). The composite process is then put in parallel with the process $EXISTS\_1$, as defined in 3.5, sharing $\{|begin\_exists\_1, end\_exists\_1, satisfied\_1|\}$. This construction (as part of the larger $VARIABLES$ ) process, is as in 3.6.

```
EXISTS_1 =
        begin_exists_1?grp : GRP?u : M →
        let
                EXISTS_(tf) =
                        ( end_exists_1!tf → EXISTS_1 )
                        □ ( satisfied_1 → EXISTS_1(true) )
        within
                EXISTS_(false)
```

Figure 3.5: The ∃ 'control' process $EXISTS\_1$

```
( chase( ( ( ( ||| u ∈ M @ (
                        [|{|begin_exists_1.grp.u, end_exists_1|grp ∈ grp|}|]
                                v ∈ M \ u @ VARIABLE_X(state_.u.v, undef) )
                ||| (
                        [|{|begin_exists_1, end_exists_1, begin_exists_2, end_exists_2|}|] v ∈ M @
                                VARIABLE_X(state_.v.v, undef)
                )
          )
          [|{|begin_exists_1, end_exists_1, satisfied_1, begin_exists_2, end_exists_2, satisfied_2|}|]
                ( EXISTS_1 ||| EXISTS_2 )
        )
        \
        {|satisfied_1, satisfied_2|}
   )
)
```

Figure 3.6: Composing the ∃ processes as part of the $VARIABLES$ process

With reference to 3.5, we see that $EXISTS\_1$, having synchronized on $begin\_exists\_1.grp.u$, $grp \in GRP$ and $u \in M$, with the $VARIABLE_X(X, x)$s, will then be prepared to engage in any number of the $satisfied\_1$ events before it engages in an $end\_exists\_1.tf$ event, $tf \in \{true, false\}$ (again synchronized with the $VARIABLE_X(X, x)$s). Note that $EXISTS\_1$ does not insist on engaging in an $end\_exists\_1.tf$ event after a $satisfied\_1$ event, if any - as that would cause deadlock if there were more than one $VARIABLE\_X(X, x)$ insisting on engaging in $satisfied\_1$ before an $end\_exists\_1.tf$.

The $EXISTS\_1$ process is prepared to engage in $end\_exists\_1.true$ if and only if it has synchronized on at least one $satisfied\_1$ event

with a $VARIABLE\_X(X, x)$ process, otherwise it will engage in $end\_exists\_1.false$. On the other hand, each $VARIABLE\_X(X, x)$ is prepared to engage in either $end\_exists\_1.true$ or $end\_exists\_1.true$, whichever is insisted on by $EXISTS\_1$.

The net effect of this is that subsequent to $begin\_exists\_1.grp.u$, zero or more $satisfied\_1$ events will occur. Then an $end\_exists\_1.tf$ event will occur, with $tf = true$ if and only if at least one $satisfied\_1$ event occurred, i.e. there exists a variable $X \equiv state_{u.v}$ satisfying the predicate '$v \in grp$ *and* $state_{u,v} \neq init$', otherwise $tf = false$.

Lastly, we optimize the process by hiding and 'chasing' all the $satisfied\_1$ events. The compression operator 'chase' selects just one path of $\tau$ (hidden) events between any two visible events, discarding all other $\tau$ paths between the visible events. The effect here is that the order in which the $VARIABLE_X(X, x)$s engage in the $satisfied\_1$ events (if they do so at all) is fixed, thereby cutting down the number of possible transitions (and states) in the LTS. This is permissible as we do not care about the order in which the $VARIABLE_X(X, x)$s engage in the $satisfied\_1$ events, only that they each get a chance to do so before the (visible) $end\_exists\_1.tf$ event occurs. The process $TH\_H$'s interaction with the $begin\_exists\_1$ and $end\_exists\_1$ events is described in 3.9.

## 3.9 $CSP_M$ model of the Trusted Host state-transition machine

The definition of $TH_H$ in chapter 2 is of a probabilistic state-transition machine (realizable by a probabilistic Turing machine). The machine is defined by its state variables, as detailed in 3.4, and its state-transitions.

The $CSP_M$ process corresponding to $TH_H$, $TH\_H$, was written to correlate as closely as possible with the original machine as defined in chapter 2. There are only two noteworthy differences, as follows.

First, state-transitions in $TH\_H$ are realized by synchronizing on $set\_$ events. For example, '$ses_{u.sid.grp} \leftarrow init$' becomes '$set\_.ses\_.u.sid.grp.init \rightarrow$' in $TH\_H$, and the current value of variable $ses_{u,sid,grp}$, for example, would be obtained in $TH\_H$ by synchronizing on a $get\_$ event of the form '$get\_.ses\_.u.sid.grp?ses\_u\_sid\_grp$'. The $\{|get\_, set\_|\}$ events are shared with the $VARIABLES$ process, which is put in parallel with $TH\_H$ (it is the

former process that actually records the current values of the state variables, $TH\_H$ itself is stateless). In a similar vein, the statement:

**for all** $v \in grp$ **do**
    $key_{v,sid,grp} \leftarrow$ *key #... and assign it (key) to all parties*
**end for**

becomes '$SET\_key\_.sid.grp.key \rightarrow$' in $TH\_H$.

Second, existential and universal quantification over the $TH\_H$ variables is not implicit as in chapter 2, rather it is realized by synchronization over the special *begin_exists_x* and *end_exists_x* channels shared with the $VARIABLES$ process. For example, the predicate '$\exists v \in grp : state_{u,v} \neq init$' is coded in $TH\_H$ as follows:

– *Does there exist a v in grp such that state_u,v != init?*
$begin\_exists\_1!grp!u \rightarrow$
$end\_exists_1?tf \rightarrow$
– *Answer in tf*

It is $VARIABLES$ that actually processes the quantification, the (boolean-valued) answer being communicated to $TH\_H$ via the *end_exists_x* channel. (In the above example, the answer to the '$\exists v \in grp : state_{u,v} \neq init$' predicate is returned in $tf$.)

The $\mathrm{TH}_H$ transitions in chapter 2 are described in a simple language similar to that proposed in [38]. Each transition starts with an input $p?m$, where $p$ is an input port (or input channel) of $\mathrm{TH}_H$. The transitions are guarded by **enabled if:** conditionals. Provided the condition is met, and the message $m$ is in the format of the enabled transition, then the machine may do a number of **outputs**, and undergo a number of changes in state variables.

In the $\mathrm{CSP}_M$, these transitions are coded naturally as a number of environmental choices over the input channels. The **enabled if:** conditionals are enforced either by guards, or by initial events. For example, in the first of the 'initialization' transitions, below, the **enabled if:** condition in chapter 2 is that '$state_{u,u} = undef$'; in the $\mathrm{CSP}_M$ this condition is ensured simply by the initial event '$get\_.state\_?u!u!undef$', which could not occur unless $state_{u,u}$ was $undef$.

Below are the two 'initialisation' transitions as coded in $TH\_H$:

```
( get_.state_?u!u!undef ->
```

```
     in.u.init ->
     -- state.u.u <- wait
     set_.state_.u.u.wait ->
     -- output:
     out_sim.u.init ->
     TH_H
)

[] ( get_.state_?u!u?state_u_u:not_corrupted ->
     in_sim.u.initialized?v:M ->
     get_.state_!v!v?state_v_v ->
     if (state_v_v==undef and not member(v,A)) or ((u==v) and state_u_u!=wait) then
        -- ignore
        TH_H
     else (
        -- state.u.v <- init
        set_.state_.u.v.init ->
        -- output:
        out.u.initialized.v ->
        TH_H
     )
)
```

Below are the two 'group key establishment' transitions as coded in $TH\_H$:

```
[] ( get_.state_?u!u?state_u_u:not_corrupted ->
     in.u.new?sid:SID?grp:GRP?(sid',grp') ->
     get_.ses_.u.sid.grp?ses_u_sid_grp ->
     get_.ses_.u.sid'.grp'?ses_u_sid_prime_grp' ->

     -- Does there exist a v in grp such that state_u,v != init?
     begin_exists_1!grp!u ->
     end_exists_1?tf ->
     -- Answer in tf

     if (not member(u,grp)) or
        (card(grp)<2) or
        (tf==true) or
        (ses_u_sid_grp!=undef) or
        (present((sid',grp')) and member(u,grp') and (ses_u_sid_prime_grp'!=finished)) then (
        -- ignore
        TH )
     else (
        -- ses.u.sid.grp <- init
        set_.ses_.u.sid.grp.init ->
```

```
      if present((sid',grp')) then (
         -- prev.u.sid.grp <- (sid',grp')
         set_.prev_.u.sid.grp.(sid',grp') ->
         -- output:
         out_sim.u.new.sid.grp.(sid',grp') ->
         TH
      )
      else (
         -- output:
         out_sim.u.new.sid.grp.(sid',grp') ->
         TH
      )
   )
)


[] ( get_!state_?u!u?state_u_u:not_corrupted ->
   in_sim.u.finish?sid:SID?grp:GRP?key_u_sim ->

   -- Does there exist a v in grp such that state_v,v == corrupted or v in A?
   begin_exists_2!grp!A ->
   end_exists_2?tf2 ->
   -- Answer in tf2

   -- Do there exist v0,v1 in grp such that ses_v0,sid,grp!=undef and
   -- ses_v1,sid,grp!=undef and prev_v0,sid,grp!=prev_v1,sid,grp?
   begin_exists_3!sid!grp ->
   end_exists_3?tf3 ->
   -- Answer in tf3

   -- Is it true that for all v in grp, ses_v,sid,grp!=finished?
   begin_forall_1!sid!grp ->
   end_forall_1?tf4 ->
   -- Answer in tf4

   get_.ses_.u.sid.grp?ses_u_sid_grp ->

   if (ses_u_sid_grp!=init) then (
      -- ignore
      TH_H
   )
   else (
      if present_(key_u_sim) and
         (tf2 or tf3) then ( -- Corrupted or inconsistent session so...
```

```
   -- output:

   -- Give key to user, use session key provided by adversary
   out.u.key.sid.grp.key_u_sim ->

   -- and delete it locally to enable forward secrecy

   -- key_u,sid,grp <- undef
   set_.key_.u.sid.grp.undef_key ->

   -- ses_u,sid,grp <- finished
   set_.ses_.u.sid.grp.finished ->

   TH_H
)
else if tf4 then ( -- First to finish (ideal) session

   -- Generate new (random) session key...
   keyR?sk ->

   -- ... and assign it to all parties of grp
   SET_key_.sid.grp.sk ->

   -- output:

   -- Give key to user...
   out.u.key.sid.grp.sk ->

   -- and delete it locally to enable forward secrecy

   -- key_u,sid,grp <- undef
   set_.key_.u.sid.grp.undef_key ->

   -- ses_u,sid,grp <- finished
   set_.ses_.u.sid.grp.finished ->

   TH_H
)
else (

   -- output:

   get_.key_.u.sid.grp?key_u_sid_grp ->

   -- Give key to user...
```

```
            out.u.key.sid.grp.key_u_sid_grp  ->

            -- and delete it locally to enable forward secrecy

            -- key_u,sid,grp <- undef
            set_.key_.u.sid.grp.undef_key ->

            -- ses_u,sid,grp <- finished
            set_.ses_.u.sid.grp.finished ->

            TH_H
        )
    )
)
```

Below are the three 'corruptions' transitions as coded in $TH\_H$. Note that the first of these transitions differs from the original description with respect to the $corOut_{sim,u}$ output. In the original state machine definition, $corOut_{sim,u}$ outputs an encoding of *all* the variables indexed by $u$ and the possible $sid$ and $grp$ with the proviso that $ses_{u,sid,grp} \neq undef$. In our case, a single $sid$ and $key$ are chosen arbitrarily. If, for that $sid$ and $key$, $ses_{u,sid,grp} \neq undef$, then $TH\_H$ offers to engage in $corOut\_sim.u.sid.grp?\_$. The extensions of $corOut\_sim.u.sid.grp$ are keys. At the same time, the $VARIABLES$ process is prepared to offer any $corOut\_sim.u'.sid'.grp'.key_{u',sid',grp'}$. Hence, when the two processes are put in parallel, sharing $corOut\_sim$ events, the effect is that the output over $corOut\_sim$ is the $corOut\_sim.u.sid.grp.key_{u,sid,grp}$ event for the arbitrarily chosen $sid$ and $grp$. This is a cheap way of outputting the $key$ data of the original encoding. It would be quite straightforward to extend this to output all of the original encoding as separate events, perhaps between 'marker' events (like $begin\_exists$ and $end\_exists$). We do not here since the $corOut\_sim$ outputs are sufficient for the properties under discussion in 3.11. They suffice because we are only concerned with the $key$ variables. Also the properties under discussion are safety properties stating that certain $corOut_{sim}$ events will not happen under certain conditions. Hence it does not matter that only one $sid$ and one $key$ is selected for each output, if a 'bad' $corOut\_sim$ event can happen, then it will be raised by FDR in a trace counter example to our assertion.

```
[] ct==adaptive & (
```

```
      get_.state_?u!u?state_u_u:not_corrupted ->
      corrupt.u.do ->
      -- state.u.u <- corrupted
      set_.state_.u.u.corrupted ->
      -- output:
      get_.ses.u?sid?grp?ses_u_sid_grp ->
      if (ses_u_sid_grp!=undef) then (
         corOut_sim.u.sid.grp?_ ->
         TH
      )
      else
         TH
)


[] ( get_.state_?u!u!corrupted ->
     in.u?any_msg ->
     -- output:
     out_sim.u!any_msg ->
     TH_H
)


[] ( get_.state_?u!u!corrupted ->
     in_sim.u?any_msg ->
     -- output:
     out.u!any_msg ->
     TH_H
)
```

## *3.10 CSP$_M$ model of the Ideal System*

The complete CSP$_M$ model of the ideal system is as in figure 3.7, below. The Trusted Host process, $TH\_H$, is as described in 3.9; the 'random' key generation process, $KEYR$, is as described in 3.7; the $VARIABLES$ process and its alphabet, $alpha\_VARIABLES$, are as described in 3.8.

$$
\begin{array}{l}
IDEAL \ = \ ( \\
\qquad ( \ TH\_H \ [|\{keyR|\}] \ KEYR(\{\}) \ ) \\
\qquad\qquad [|diff(alpha\_VARIABLES, \{|satisfied|\}|] \\
\qquad\quad VARIABLES \ ) \\
\setminus \ alpha\_VARIABLES
\end{array}
$$

Figure 3.7: The complete IDEAL system

## 3.11 $CSP_M$ transcriptions of two group key establishment properties

In chapter 2, there are cited a number of basic service and security properties that one can expect of n-party key establishment. These properties are stated in natural language terms. In this section we discuss how we may transcribe two important security properties into $\mathrm{CSP}_M$ specification processes so as to demonstrate that our model of the GKE Trusted Host exhibits those properties through refinement checks using FDR.

The two security properties that we would like to show $TH\_H$ exhibits are as follows:

- **Key Secrecy:** For uncorrupted sessions, the session key is generated randomly and secretly by $TH\_H$, the adversary never having any knowledge of the key.
- **Perfect Forward Secrecy (PFS):** The compromise of long-term session keys cannot result in the compromise of past session keys.

We now describe how the above two properties may be transcribed in turn.

### 3.11.1 Key Secrecy

This property is verified through two refinement tests.

The 'randomness' of the key generation is verified by showing that $IDEAL$ with all events except the $keyR$ events hidden is failures refined by the process $RAN\_GEN\{\}$). At any time, $RAN\_GEN(\{\})$ will either $STOP$ or else engage in an arbitrary $keyR.x$ event, with the proviso that $keyR.x$

has not occurred before in the trace. This behaviour continues whilst there are 'unused' keys, $x$, remaining. That the specification can $STOP$ at any time is necessary to account for the fact that in some behaviors of the Ideal System, it may be the case that, after a time, no new keys are generated. The $\text{CSP}_M$ for the assertion is as follows:

```
RAN_GEN(used) =

   STOP |~| ( keyR?x:diff(Keys(n),used) -> RAN_GEN(union(used,{x})) )

assert RAN_GEN({}) [F= IDEAL \ diff(alpha_TH,{|keyR|})
```

There are a number of ways that we could feasibly verify the 'secrecy' of the key generation, with varying degrees of informal argument. We shall verify the property by showing that if an $out\_u.key.sid.x$ event occurs, where $sid$ is a session identifier and $x$ a key, and if the pair $sid.x$ had been previously 'leaked' through some $corOut\_sim\_.v.sid.grp.x$ event, then it is indeed the case that $v$ is a member of $grp$. As the $corOut\_sim\_.v$ event only occurs if and only if $v$ has been corrupted, we are confirming that the only way a newly agreed key could have been compromised is through the corruption of a group member during negotiation of that key - and in that case 'all bets are off'.

For this assertion, we can exploit the fact that our model of the Ideal System is symmetrical in both the keys[8], $Keys(k)$, and the session identifiers, $SID$. This means that we can afford to verify the assertion once for some fixed key $x$, and session identifier, $sid$. If the assertion holds for these particular values, then it will hold for all keys and all session identifiers. We shall fix $x$ as $< 0, \ldots, 1 >$, and $sid$ to be 1.

### 3.11.2 PFS

The Trusted Host machine ensures this property by overwriting local $key_{u,sid,grp}$ variables when a key establishment session is finished. We may verify PFS by showing that, for uncorrupted completed sessions, the session identifier/key pair is never 'leaked' via the $corOut\_sim\_$ channel. By the

[8]This would not be the case if our key generation process had not been 'random', cf. 3.7.

same argument applied to the above 'secrecy' property, we may fix the session identifier and key.

## *3.12 Running the Scripts*

The compilable $\text{CSP}_M$ scripts for the GKE Ideal System are to be posted in the public section of the MAFTIA home-page at [48]. It is intended that all $\text{CSP}_M$ models developed under WP6 will eventually be posted on this site for download.

For any particular number of participants, $n$, a-priori uncorrupted participants, $H$, bound on key length, $k$, session identifiers, $SID$, and corruption model, $ct$, there are six .csp scripts in total that constitute the CSP model and properties of the Ideal System.

The *types.csp* script contains the datatype, nametype, function and channel definitions.

The *variables.csp* script contains the $\text{CSP}_{itM}$ processes for reading from and writing to the TH variables and existentially and universally quantifying over them.

The *TH.csp* script contains the model of the TH state-transition machine.

The *IDEAL.csp* script brings together the processes of *variables.csp* and *TH.csp* to form the complete model of the Ideal System.

The *properties.csp* script contains $\text{CSP}_{itM}$ transcriptions of the 'informal properties' that are asserted of the Ideal System.

In addition, there are a number of *test* .csp scripts. It is the *test* scripts that are loaded into FDR. The *test* scripts define the parameters $n$, $H$, $k$, $SID$, and $ct$. Each *test* script includes the other five (non-*test*) scripts. At this point the datatypes of *types.csp* are instantiated from $n$, $k$ and $SID$.

For example, in *test1.csp* we have:

```
n = 2
-- Number of participants


nametype H = {1..n-1}
-- a-priori uncorrupted participants


A = diff(M,H)
```

```
k = 2
-- Bound on key length


SID = {0..1}
-- The session identifiers


ct = static
-- The corruption model is static
```

## *3.13 Report*

The $\text{CSP}_M$ model of the Trusted Host detailed in this chapter has been compiled and 'sanity checked' for small values of $n$, $k$ and $SID$, typically: $n = 2$, no a-priori corrupted parties, $ct = static$, $k = 1$ (providing two keys), and $|SID| \leq 2$. These values are too restricted, in our opinion, to claim at this point much success in demonstrating the viability of using CSP/FDR to verify machines of the rigorous secure reactive systems theory. That said there are a number of promising avenues that we have yet to pursue that could lead to significant reductions in state space. For example, the fact that the models are symmetric in the session identifiers, $SID$, gives us scope to restrict $GRP$ (say to one group, $grp$, of each cardinality $0 \leq |grp| \leq 2^k$) without weakening our model too much.

# 4 CSP Modelling Of Selected TTCB Security Services

## *4.1 Background*

This section describes an analysis of some of the security services supported by the Trusted Timely Computing Base (TTCB). The TTCB is a security kernel composed of a number of distributed, trusted components (local TTCBs) connected via a secure, synchronous control network, see Figure 4.1 below. In particular, the TTCB is a synchronous entity both locally and across the control network. As such, the TTCB is intended as a key stone to secure systems of a larger and more general nature. Such systems, providing the TTCB remains secure (and fail-silent) may then be tolerant to intrusions and arbitrary failures.

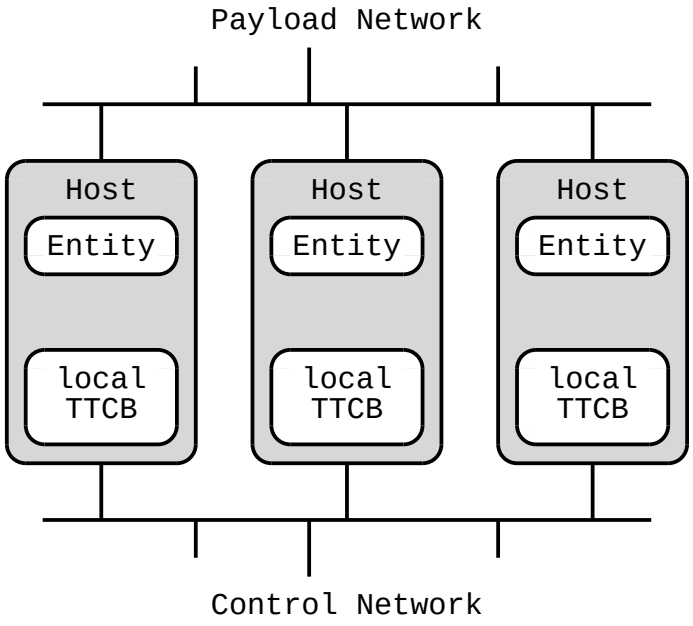


Figure 4.1: The Trusted Timely Computing Base (TTCB).

The TTCB currently supports three security services, two of which have been analysed and partially verified as part of the MAFTIA project. These are the Local Authentication (LAS) and the Trusted Block Agreement (TBA)

Services. The former may be used to establish a secure channel between an Entity (the user) and a local TTCB, while the later is used to reach consensus among a group of Entities over a pre-determined decision. The Trusted Block Agreement service itself relies on the Local Authentication Service to establish secure channels between Entities and their local TTCBs. The third and last security service is a random number generator. However, we do not present an analysis of this service here.

The Trusted Block Agreement (TBA) service has in turn been used as the basis for a reliable multicast protocol, aiming to tolerate Byzantine failures through reliance on the Trusted Block Agreement service. This protocol, the Byzantine Resilient Multicast (BRM) protocol, has been analysed in part but the modelling is as yet incomplete.

As with previous MAFTIA WP6 work, the verification and assessment of the two security services or more rather protocols was undertaken using the CSP formalism and the accompanying FDR model checker. The choice of formalism was influenced partly by the problem in hand, the verification of two security services, and partly with the intention of developing existing techniques and models.

The verification of the Local Authentication and Trusted Block Agreement services is currently partial as formal arguments extending the model checking to general cases (arbitrary numbers of local TTCBs, Entities) have not yet been proposed. However, for each associated protocol the required properties have been completely verified for a limited number of configurations (parties, session parameters). The details of this verification follow, including a brief description of each of the services.

In the analysis of a security system, the environment it inhabits and has little control over requires special attention. Building on previous work, the intended environment and system may be identified with key MAFTIA concepts and models. These include the failures model, synchrony, topology and capabilities of the adversary if present. Finally, the security goals of a system may be identified with formal properties to be verified and assessed in the context of the given environment.

### 4.1.1 The Local Authentication Service

The Local Authentication Service allows an Entity to establish a secure channel to a local TTCB. The service uses an Authenticated Key Establishment protocol to deliver a secret key to the local TTCB and in the process authenticate the TTCB to the Entity (i.e. one-way entity authentication). The protocol simply consists of a request from Entity to local TTCB followed by a reply. The request contains a secret key derived by the Entity, and a challenge, both encrypted using the local TTCBs Public Key. The reply contains a signature of the challenge, signed by the local TTCBs Private Key there by authenticating the local TTCB to the Entity who derived the challenge. It should be noted that the full protocol contains extra components to the request and reply that are not directly related to the service. A more detailed description may be found in a paper by Correia, Veríssimo and Neves [49].

The security protocol may be represented in the usual manner by a series of messages between the legitimate principals. The signature of the Challenge, $Ne$, is represented by an encryption of the hash value of the Challenge using the local TTCBs private key. Although the paper [49] does not define how the signature is produced, the representation below captures the assumptions over the signature in terms of the assumptions over Hash functions and encryption.

$$\begin{aligned} \text{Msg 1. } E \rightarrow T &: \{Ke, Ne\}_{pk(T)} \\ \text{Msg 2. } T \rightarrow E &: \{H(Ne)\}_{sk(T)} \end{aligned}$$

where $Ke$ is the shared Key, $Ne$ is the Challenge, $pk(T)$ and $sk(T)$ denote the public and secret keys of the TTCB $T$.

Each party (Entity and local TTCB) communicates over a payload network, an open asynchronous network. Any message may be read, modified, reordered, delayed or replayed. In particular, messages are allowed eventual delivery. While the local TTCB must subscribe to a strictly controlled failure model (fail-silent), Entities may fail arbitrarily. In terms of topology the network may be considered a bijection between Entities and local TTCBs as local TTCBs and correct Entities act independently. For the purposes of this analysis, the familiar Dolev-Yao model (strong encryption, strong attacker, strongly typed messages) defines the assumed capabilities of a malicious agent

either acting as a legitimate Entity or as an external agent.

The Local Authentication Service specifies the following security goals or properties that should hold after a given protocol run. These are:

- *SK1. Implicit Key Authentication.* The Entity and the TTCB know that no other entity has the key.
- *SK2. Key Confirmation.* Both the Entity and the TTCB know that the other has the key.
- *SK3. Authentication.* The entity has to authenticate the local TTCB.
- *SK4. Trusted Against Known-Key Attacks.* Compromise of past keys does *not* allow either (1) a passive adversary to compromise future keys, or (2) impersonation by an active adversary.

### 4.1.2 The Trusted Block Agreement Service

The Trusted Block Agreement Service allows a group of Entities to reach consensus over a decision, each participates by proposing values which are then input to a chosen decision function to determine the decision. Entities communicate over secure channels with their local TTCB, established previously by using the Local Authentication Service. There are two stages to the service protocol, a *propose* stage followed by a *decide* stage.

The propose stage must finish before a given 'propose by' time *tstart*, Entities may or may not propose depending on their preference (and of course the environment.) During the decide stage Entities may request the agreed decision. The decide stage begins as soon as a decision may be reliably released. At the very least, the decide stage begins a fixed duration of time after the propose stage has finished. It may also begin when a local TTCB holds a complete set of proposals for all Entities in the group.

Each local TTCB communicates with all other local TTCBs via a closed, synchronous control network. The network has well defined properties, in particular:

- security (confidentiality, integrity and authentication),
- (upper) bounds on message latencies,

- (upper) bounds on message omissions.

These properties are the Abstract Network properties and may be found on page 4 of [49]. In short, the network provides a reliable and timely medium. Although under normal circumstances a local TTCB may fail (silently), the protocol assumes that all local TTCB function perfectly. The case in which a local TTCB may fail (by crashing) is covered by an extended version of the paper [49]. As before Entities may fail arbitrarily. As the network is already assumed secure, any malicious activity beyond traffic analysis must occur at the local TTCB user interface.

The Trusted Block Agreement Service specifies the following security goals or properties that should hold after a given protocol run. These are:

- *AS1: Termination.* Every correct entity eventually decides a result.
- *AS2: Integrity.* Every correct entity decides at most one result.
- *AS3: Agreement.* If a correct entity decides a *result*, then all correct entities eventually decide result.
- *AS4: Validity.* If a correct entity decides *result* then *result* is obtained applying the function *decision* to the values proposed.
- *AS5: Timeliness.* Given an instant *tstart* and a known constant $T_{agreement}$, a process can decide by $tstart + T_{agreement}$.[1]

## *4.2 Datatypes*

The modelling of the Local Authentication and Trusted Block Agreement services began with the basic set of MAFTIA models and datatypes used for the Synchronous Contract-Signing Protocol CSP verification presented in Deliverable D7 [3]. Wherever possible the $\text{CSP}_M$ scripts adopt the LAS and TBA protocols original naming conventions from [49].

For each of the protocols, a given identity may be assigned information such as keys, challenges and so on. A datatype Principal defines the identifiers, while the subsets $TTCBs$ and $ENTs$ define the particular subset of

[1] The TTCB is a timely component in a payload system with uncertain timeliness. Hence, the Timeliness property is valid only at the TTCB interface. An Entity can only decide with the timeliness the payload network permits.

Principal considered for a given analysis. Finally, the subset $P$ is constructed from the union of $TTCBs$ and $ENTs$.

```
datatype Principal = TTCB | TTCB1 | TTCB2 | Entity | Entity1 | Entity2
-- The principals involved:
-- TTCB, TTCB1, TTCB2 - Trusted local TTCBs.
-- Entity, Entity1, Entity2 - Entities intending to use a TTCB service.

nametype ENTs = {Entity,Entity1}
nametype TTCBs = {TTCB}
-- ENTs is the set of entities being considered by an analysis.
-- TTCBs is the set of TTCBs being considered (they form the distributed kernel).

nametype P = Union({ENTs,TTCBs})
-- all the parties involved in the authentication, and capable
-- of sending/receiving messages. Note that the Spy is absent
-- as his behaviour is intimately tied to the network model.

otherthan(this) = diff(P,{this})
-- an entity other than 'this' one.
```

All principals are associated with a unique, finite set of keys and challenges to use during protocol runs. These are the only secrets used during protocol runs, all other values may be considered public knowledge. Principals are also associated with a Public Key, required for the LAS protocol and distributed reliably to all other principals. Both keys and challenges are defined as integer range types. To avoid confusion, public keys are distinct from shared keys. Private keys corresponding to public keys are not currently modelled explicitly.

```
nK = 2
nametype Key = {1..(1+nK)*(card(Principal))}
-- Public keys and shared keys are taken to be chosen from *disjoint* finite sets.
-- There is one public key and nK unique keys to share for each principal.

nC = 2
nametype Challenge = {1..nC*(card(Principal))}
-- All challenges are taken from a finite set. There are nC challenges
-- allocated to each principal. Challenges are unique to each run of the
-- authentication protocol. Thus card(C) is the maximum number of times
-- the authentication protocol can be run in our model.
```

For the TBA and BRM protocols, the identity of an Entity is important so a corresponding entity identification type (Entity Id) is introduced. The identity of an Entity is established by a local TTCB during the Local Authentication Service. An Entity receives a fresh identity for every call to the

service, which will be associated by the local TTCB with the secret key established. Entity Ids are therefore uniquely associated with local TTCBs and again defined as an integer range type. Note that the definition is relative to the subset of local TTCBs considered.

```
nE = 3
nametype Eid = {1..nE*card(TTCBs)}
-- the set of Entity Identification numbers.
-- There are nE entity ids assigned to each TTCB.
```

The TBA and BRM protocols also deal with static groups of Entities, represented as lists of Entity Ids. For instance, the group of Entities involved in a given agreement for the TBA and the group involved in a multicast for the BRM. In the case of the TBA, the group defines which proposals a local TTCB will accept, and determines when all proposals have been received. In the case of the BRM, the group defines the recipients for an Entity to multicast and determines when all parties have acknowledged receipt of a correct message. Entity lists are defined symbolically, a $\text{CSP}_M$ function *MyGroup* then maps list names to an an associated list of Entities (Entity Ids). Currently, there are only two Entity lists.

```
datatype List = ElistA | ElistB

nametype Elist = {ElistA}
-- The datatype for 'elist' - a symbolic list of Entities.
-- There is currently only one group of ENTs, ElistA.
```

Finally, various other datatypes are required for the LAS, TBA and BRM protocols. In particular, both the TBA and BRM protocols take a timestamp *tstart* and a 'data' value to be proposed and decided. Data values are treated data-independently by the TBA and BRM, and tested for equality only by the BRM. The timestamp *tstart* is integral to the TBA but data-independent to the BRM protocol.

```
nametype Protection = {1..1}
-- the protection parameter is provided in a Local Authentication Call to
-- choose the protection of the secure channel (aka eid & Ket) being established.
-- Ranges:
-- {1..1}: Authenticity only required:
-- {2..2}: Authenticity and Integrity only required:
-- {3..3}: Authenticity, Integrity, Confidentiality:

nametype TStart = {5}
-- only one tstart time for the TBA agreements.

nametype Decision = {1..1}
-- only one decision function.
```

```
nametype Value = {1..2}
-- decision values are represented by number ranges.
```

For each of the datatypes above various CSP$_M$ functions are defined to select keys, challenges and so on for a particular principal, Entity or local TTCB. Principals are given an arbitrary order to determine the assignment. There are other useful helper functions but these are omitted for lack of space.

```
P_Order = <TTCB,TTCB1,TTCB2,Entity,Entity1,Entity2>
-- The sequence, P_Order, imposes an arbitrary order on the datatype
-- Principal.

E_Order = filter(P_Order,ENTs)
-- An ENTs ordering.

T_Order = filter(P_Order,TTCBs)
-- A TTCBs ordering.

Public_Key(p) = index(p,P_Order)
-- The PKI repository.
-- Retrieve a public key: Public_Key : P -> Key

MyChallenges(p) = let i = index(p,P_Order) within {1+(i-1)*nC..i*nC}
-- MyChallenges
-- The set of unique Challenges used by a Principal, p.
-- Note that nC is the number of challenges per principal.

MyKeys(p) = let i = index(p,P_Order) within {#P_Order + k | k<-{1+(i-1)*nK..i*nK}}
-- MyKeys
-- The set of unique Keys to share used by a Principal, p.
-- Note that nK is the number of keys per principal (>=1).

FirstEid(t) = let i = index(t,T_Order) within 1+(i-1)*nE
LastEid(t) = let i = index(t,T_Order) within i*nE
NextEid(t,eid) = eid+1

MyEids(t) = {FirstEid(t)..LastEid(t)}
-- MyEids
-- The set of unique entity ids used by a TTCB, t.
-- Note that nE is the number of eids per ttcb.

MyGroup(ElistA) = <FirstEid(t) | t<-T_Order>
-- MyGroup
-- Give a meaning to Entity Id Lists.
```

## *4.3 The Local Authentication Service*

The modelling of the LAS protocol is decomposed into the roles of the Entity, local TTCB and the payload network. As with previous MAFTIA work, the payload network ($PAYLOAD$) combines a malicious entity known as the Spy ($SPY$). A full protocol run then encorporates an Entity ($AKE\_ENTITY$) and local TTCB ($AKE\_TTCB$) brought together over the payload network.

The behaviours of both the Entity and local TTCB are represented by simple finite state machines. An Entity initiates the protocol run by a request, and then waits for a valid reply before committing to the protocol run while the TTCB waits for a valid request before committing and replying. As such, the protocol itself is fairly simple and it is the payload network that holds the real complexity of the model.

### 4.3.1 The Payload Network

The datatype $MSG$ recursively defines an infinite strongly-typed message datatype.

```
nametype Data = Union({Key,Challenge,Protection,Eid})
-- a union of all message component types, to be used in a sequence.

nametype Text = { <a>,<a,b>,<a,b,c> | a<-Data, b<-Data, c<-Data }
-- actual messages are sequences of Data (keys, challenges, protections, eids).
-- Messages have up to three components. card(Text) = 5+5^2+5^3.

datatype MSG = C.MSG.MSG | E.Key.MSG | S.P.MSG | T.Text

-- A 'universal', recursively defined infinite set that
-- stipulates the pattern that 'message' strings are to
-- take. These 'messages' strings are the data conveyed
-- over the network channels.
```

A typical message string has the form $E.Ku.T.\langle Ke, protect, Ne\rangle$ where: $Ku$, $Ke$ are keys, *protect* is a protection parameter and $Ne$ is a challenge. This is the message form used by the LAS request, the reply by the local TTCB is of the form $C.S.TTCB1.\langle Ne\rangle.T.\langle eid\rangle$ where: $TTCB1$ is a TTCB identifier, $Ne$ is a challenge and *eid* is an Entity Id.

The type constructor C represents the concatenation of two messages. The type constructor E represents encryption by a valid key, Key. The type constructor S represents the creation of the cryptographic signature of a message by a principal in P (assumed to use his private key). Finally the type constructor T represents a plain-text message composed of up to three components, each of which may be a Key, Challenge, Protection parameter or Eid.

In finite-state model checking, we must restrict our models to finite datatypes. For the purposes of the Local Authentication Service model, a restriction is imposed on the infinite type $MSG$ to produce a finite type $msg$. The restriction considers all messages that a correct party may be induced to release, all deductions the Spy may make and any initial knowledge he may possess. In this way, the restriction includes all the messages that could possibly pertain to the session in question.

The restriction is defined as follows. $ENT\_M(e,t)$ defines the set of messages an Entity $e$ may be induced into sending to a TTCB $t$ while $TTCB\_M(t,e)$ defines the set of messages a TTCB $t$ may be induced into sending to an Entity $e$. Encrypt and Signature are functions that construct encryptions and signatures. Note that Entities have a perfect knowledge of a local TTCBs Public Key, and TTCBs do not use their Challenges (hence they cannot be learnt by a Spy).

```
ENT_M(e,t) = {Encrypt(Public_Key(t),(T.<k,p,c>))
                  | k<-MyKeys(e), p<-Protection, c<-MyChallenges(e)}
-- The Entity may choose to send (to a TTCB t):
-- E.ku.T.<k,p,c>
--         where, ku == PublicKey(t), <k,p,c> in his possession.

TTCB_M(t,e) = {C.Signature(t,T.<c>).T.<eid>
                  | ent <- ENTs, c<-MyChallenges(ent), eid<-MyEids(t)}
-- The TTCB may choose to send (to e):
-- C.S.t.T.<c>.T.<eid>
--         where, <c,eid> of c of correct type and eid of MyEids(a).
-- Note that the TTCB does not know about MyChallenges therefore
-- he may be duped into accepting any entities Challenge!
```

The set $correct\_msg$ defines the set of messages that $CORRECT$ Entities or (trusted) TTCBs may send willingly. The set $CORRECT$, a subset of ENTs, is introduced as the Spy is allowed a statically defined set of $CORRUPT$ Entities from which to base future attacks. This capability is needed for the informal security property SK4 (over Known-Key Attacks).

```
nametype CORRUPT = {Entity1}
```

```
nametype CORRECT = diff(ENTs,CORRUPT)

correct_msg = Union({ENT_M(c,t),TTCB_M(t,e) | c<-CORRECT, e<-ENTs, t<-TTCBs })
```

The payload datatype, msg, is then defined as follows.

```
msg = Union({correct_msg,
             InitialKnowledge,
             deductions(m0) | m0 <- correct_msg})
```

Note that it is an assumption that the deductions function distributes over sets of messages, i.e.:

$$Deductions(X \cup Y) = Deductions(X) \cup Deductions(Y)$$

$InitialKnowledge$ is defined as the set of all messages that may be constructed from the initial knowledge of the Spy, and be accepted by some $CORRECT$ Entity or (trusted) local TTCB. As such it is maximal with respect to the deductions the Spy may make. The Spy will possess the keys and challenges of all $CORRUPT$ Entities. Unfortunately, the Spy is restricted to messages accepted by local TTCBs as $CORRECT$ Entities only accept messages containing signatures by TTCBs. Note that $InitialKnowledge$ includes all $ENT\_M(e : CORRUPT, t)$ and all mismatches between Keys and Challenges.

```
InitialKnowledge = {Encrypt(Public_Key(t),(T.<k,p,c>))
                       | t<-TTCBs, e1<-CORRUPT, e2<-CORRUPT, k<-MyKeys(e1),
                           p<-Protection, c<-MyChallenges(e2)}
```

We then define the standard network channels, send and receive:

```
channel send : P.P.msg
channel receive : P.P.msg
```

A simple asynchronous network medium, MEDIUM, is used to model the sending and receipt of a message in the face of the Spy. A message may only be received after it has been sent by a $CORRECT$ entity or local TTCB. As soon as the Spy overhears the message, it can replay any message deduced to any principal in P. If the overheard message contains an accessible key (given the Spy's initial knowledge) then the key may be leaked. Note the use of 'otherthan' to exclude entities sending and receiving messages from themselves.

```
-- a 'debug' deduce channel to signal deductions.
channel deduce : msg

MEDIUM(message,sent) =
```

```
  -- accept the sending of the message & update 'sent'.
  (send?p1:union(CORRECT,TTCBs)?_:otherthan(p1)!message -> MEDIUM(message,true))
[]
  -- if sent, then may be received.
  (sent & receive?p1?_:otherthan(p1)!message -> MEDIUM(message,sent))
[]
  -- if sent & possible to deduce something, then may be received.
  (sent & [] m:deductions(message) @ deduce.m ->
        receive?p1?_:otherthan(p1)!(m) -> MEDIUM(message,sent))
[]
  -- if sent and (simply!) contains a key, then Spy may 'leak' it.
  ((sent and hasKey(message)) & leak.GetKey(message) -> MEDIUM(message,sent))
```

Although the Spy, given his initial knowledge, is capable of many simple one-message deductions the current analysis has not found any deductions that are actually possible from the limited set *correct_msg* and accepted by a correct party ($union(CORRECT, TTCBs)$). This is partly due to the simplicity of the protocol (two messages, PKI assumed), its assumed operating environment and the restriction to only one-message deductions. As a consequence, the ability of the Spy to capture keys is similarly non-existent here. It should be noted that the Entity Id is returned as plain-text. While this does not affect the properties of the protocol it does open a door to future issues, to be discussed in more detail in section 4.3.6.

Finally, the payload network $PAYLOAD$ is given below. The process $SPY$ captures the behaviour of the Spy relative to just its initial knowledge.

```
SPY(Initial) = [] message:Initial @ receive?p1?_:otherthan(p1)!message ->
                                                              SPY(Initial)

PAYLOAD = (||| message : correct_msg @ MEDIUM(message,false))
               ||| SPY(InitialKnowledge)

alpha_PAYLOAD = {| deduce,send,receive |}
```

### 4.3.2 The Entity and Local TTCB

The full Local Authentication Service protocol may be summarised by the modified message exchange below.

$$\begin{aligned} \text{Msg } 1.\ E \rightarrow T &: \{Ke, protect, Ne\}_{pk(T)} \\ \text{Msg } 2.\ T \rightarrow E &: \{H(Ne)\}_{sk(T)}.entity\_id \end{aligned}$$

where in addition, *protect* is the Protection parameter and *entity_id* is the Entity Id chosen by the TTCB $T$.

When the Entity initiates the protocol run it chooses a shared key, a protection parameter, a challenge and a local TTCB to provide the service. In return, for every valid request the local TTCB receives it replies with an Entity Id as well as its signature of the challenge. The (public) Entity Id is then used in later services such as the TBA and BRM. Note that the Entity Id is not protected when it is returned to the Entity. Although this has no bearing on the protocol at hand it clearly has implications for future interactions. This issue will be discussed in more detail later on in section 4.3.6.

The behaviour of the Entity and local TTCB are modelled by the CSP processes $AKE\_ENTITY$ and $AKE\_TTCB$, where the prefix 'AKE' is an abbreviation for Authenticated Key Establishment. Each process is parameterised by the choices to be made over the shared key, challenges, local TTCB, entity id and so on, as these choices do not depend on previous messages. The process $AKE\_ENTITY$ takes the parameters *this*, *ttcb*, *key*, *protect* and *c*. *this* is the identifier of the Entity; *ttcb* is the identifier of the local TTCB; *key* is the shared Key; *protect* is the Protection parameter and *c* is the Challenge. The process $AKE\_TTCB$ takes the parameters *this* and *eid*. Again, *this* is the identifier of the local TTCB and *eid* is the Entity Id.

Both processes are implemented as (almost) a-cyclic state-machines, the only cycles being transitions to self on receiving junk messages. Each process has broadly aligned states, Start, Request, Reply, Accept and Reject. The Reply state is extended by the information received in the request while the Accept state is extended by the Entity Id received in the reply. Entities progress from Start, to Request, to Accept while local TTCBs progress from Start, to Reply, to Accept.

```
datatype AKE_state = Start | Request | Reply.P.Key.Protection.Challenge.Eid |
                     Accept.Eid | Reject
```

With hindsight the reflexive cycles are strictly unnecessary as junk messages do not affect either the behaviour of the Entity or the Spy. In terms of data-independence methods, the implicit equality test introduced by a constrained input is allowable as long we can guarantee that the result of the equality test proving false will never result in the process performing a trace that it could not have performed were the test to prove true. This is a form of the condition **PosConjEqT** (Positive Conjunctions, see [50, 51]).

To initiate and conclude the protocol run, the Entity synchronises on user-interface input and output events (see user.cspm). For instance, the event

$$input.Entity.local\_auth.TTCB.17.1.9$$

denotes a request from Entity to initiate a protocol run with TTCB using the key 17, the protection parameter 1 and the challenge 9. Finally, both the Entity and local TTCB use a signal channel to make assertions about the state of the protocol at defined positions through a run (see section 4.3.3).

```
-- USER is a set of ENTs that are allowed to initiate requests
-- (use input/output), and all TTCBs.
USER = union(CORRECT,TTCBs)

-- The Entity Input-Output Channels --
challenge_signature = { Signature(t,T.<x>) | t<-TTCBs, x<-Challenge }

-- Service Requests, i.e. the API call to the local TTCB.datatype
channel input : USER.local_auth.TTCBs.Key.Protection.Challenge

-- Service Replies.
channel output : USER.local_auth_return.Eid.challenge_signature
```

The $\text{CSP}_M$ $AKE\_ENTITY$ process is as follows.

```
-- AKE_ENTITY:
-- For a given protocol run, the parameters ttcb,key,c,eid
-- should define a unique tuple over all protocol runs.
-- The entity owns key and c, but receives eid from the ttcb.

AKE_ENTITY(this,ttcb,key,protect,c) =
  let
    -- common to protocol run.
    Ku = Public_Key(ttcb)
    request = Encrypt(Ku,T.<key,protect,c>)
    Valid_reply = {C.Signature(ttcb,T.<c>).T.<e> | e <- Eid}
    GetEid(C.(m).T.<e>) = e

    -- the actual protocol transitions.
    PROTOCOL1(state) =

    -- the Start state...
    state == Start & (

        -- handle junk messages before we send our request.
        (receive.this?p:otherthan(this)?_ -> PROTOCOL1(state))
```

```
      []
        (
        -- make a request to the TTCB, encrypting the input parameters.
        send.this.ttcb.request ->
        -- transition to an intermediate request state.
        PROTOCOL1(Request))
    )

    -- the Request state...
    [] state == Request & (

      -- take a reply, any reply.
      receive.this?p:otherthan(this)?reply ->
        if member(reply,Valid_reply)
          -- when a correct reply, transition to an accept state.
          then PROTOCOL1(Accept.GetEid(reply))
          -- handle any junk messages and continue listening.
          else PROTOCOL1(state)
    )

    -- the Accept state...
    -- The state includes eid for benefit of the entity.
    [] ([] eid:Eid, state==(Accept.eid) @

         -- just before we finish, assert the security properties.
        -- SK1: The key is secret to just this entity and the ttcb...
        signal.Claim_Secret.this.ttcb.key ->
        -- SK2 & SK3: (given ttcb.Running) The ttcb has the correct key for
        -- this particular run of the protocol (unique by c,eid??).
        signal.Commit.this.ttcb.<key,c> ->
        -- output the results for the user.
        output.this.local_auth_return!eid!S.ttcb.T.<c> ->
        -- the protocol ends...
        SKIP
    )
  within
    input!this!(local_auth.ttcb.key.protect.c) ->
        PROTOCOL1(Start)
```

The $\text{CSP}_M$ $AKE\_TTCB$ process is as follows.

```
-- AKE_TTCB:
-- The AKE_TTCB protocol run is instantiated with the Entity id the TTCB
-- will choose. The TTCB runs the protocol below and then recurses to the
-- next Entity id or just SKIP. The choice at each stage could be converted
-- to 'choice from a set'.
```

```
AKE_TTCB(this,eid) =
  let
    -- common to protocol run.
    Ku = Public_Key(this)
    Decide(E.k1.T.<k2,p,c>) = k1 == Ku and member(k2,Key)
                                  and member(p,Protection)
                                  and member(c,Challenge)
    Decide(_) = false
    Run(E.ku.T.<k,p,c>,eid) = <k,c>
    Params(E.ku.T.<k,p,c>) = k.p.c

    -- the actual protocol transitions.
    PROTOCOL1(state) =

      -- the Start state...
      state == Start & (

      -- Take a message, any message.
      receive.this?entity:otherthan(this)?message ->
        -- does the message look correct?
        if (member(entity,ENTs) and Decide(message))
          -- if so, proceed and handle the request.
          then
            -- decide on an entity id & signal running.
           signal.Running.this.entity.Run(message,eid) ->
                  -- transition to the reply state...
                PROTOCOL1(Reply.entity.Params(message).eid)
          -- if not, reject and try again.
         else PROTOCOL1(state)
    )

    -- the Reply state...
    [] ([] entity:ENTs,key:Key,protect:Protection,c:Challenge,eid:Eid,
                state==(Reply.entity.key.protect.c.eid) @

      -- handle junk messages before we send our reply.
      (receive.this?p:otherthan(this)?_ -> PROTOCOL1(state))
    []
      (-- send the signed challenge and entity id, eid.
      send.this.entity.C.Signature(this,T.<c>).T.<eid> ->
        -- transition to an accept state..
        PROTOCOL1(Accept.eid))
    )
```

```
    -- the Accept state...
    -- The state includes eid for benefit of the entity.
    [] ([] eid:Eid, state==(Accept.eid) @

        -- the protocol ends...
        SKIP
    )
  within
    PROTOCOL1(Start)
```

An Entity may perform any number of protocol runs, each using a fresh Challenge (and presumably a fresh key) until the finite set is exhausted. This behaviour is captured by a control process, $ENTITY\_$, that regulates the input events. The full behaviour of an Entity, $ENTITY$, is then the parallel composition of the control process, $ENTITY\_$, and all possible $AKE\_ENTITY$ processes within the scope of the analysis, i.e. the number of keys, challenges, local TTCBs. Note that under this model an Entity may use the same key twice.

```
ENTITY_(this,{}) = SKIP
ENTITY_(this,ChallengesRemaining) =
        input.this.local_auth?ttcb?k:MyKeys(this)?protect?c:ChallengesRemaining ->
                ENTITY_(this,diff(ChallengesRemaining,{c}))

ENTITY(this) =
        (ENTITY_(this,MyChallenges(this)))
          [| {|input.this.local_auth|} |]
        (||| ttcb:TTCBs,k:MyKeys(this),protect:Protection,c:MyChallenges(this) @
                                AKE_ENTITY(this,ttcb,k,protect,c))
```

A local TTCB may also receive any number of protocol runs, each using a fresh $Eid$ until the finite set is exhausted. While Entity requests must run in parallel the local TTCB replies may be allowed to run sequentially. This is demonstrated by the observation that the (atomic) sequential replies may be rearranged into any (correct) order the network desires. Further, the choice of Entity Id by the local TTCB may be fixed each time as Entities (and currently the Spy) treat Entity Ids transparently. In particular, none of the security properties of the following section depend on the Entity Id.

The full behaviour of a local TTCB, $LOCALTTCB$, is then the sequential composition of all enumerated $AKE\_TTCB$ processes as above. The process $LOCALTTCB\_$ takes an identity and a list of Entity Ids to use, here the sequence of unique Entity Ids assigned to a TTCB in ascending order.

```
LOCALTTCB_(this,<>) = SKIP
```

```
LOCALTTCB_(this,<e>^eids) = AKE_TTCB(this,e); LOCALTTCB_(this,eids)

LOCALTTCB(this) = LOCALTTCB_(this,<FirstEid(this)..LastEid(this)>)
```

### 4.3.3 CSP transcription of the informal properties

As described in the introduction, the Local Authentication Service supports four security properties SK1 to SK4. Each property is expected to hold after a given protocol run has completed, some such as secrecy must also hold during the run. The major issues in transcribing the informal properties to CSP involve giving interpretations for Agreement (SK2), Authentication (SK3) and to a lesser extent Secrecy (SK1).

To aid the CSP specification of properties, both the Entity and local TTCB use a *signal* channel to make assertions about the state of the protocol at defined positions through a run. All properties only make assertions over the traces of the protocol model. The *signal* (and *leak*) channel is defined as follows:

```
nametype Secrets = Key

channel leak : Secrets
-- This is the special leak channel over which the Spy
-- may leak secrets at his discretion (the Spy owns leak).

-- LAS: the three types of signal we need: 'secrecy', and 'commitment' to the
--       other 'running'.

datatype Signal = Claim_Secret.P.P.Secrets | Commit.P.P.Text | Running.P.P.Text

channel signal : Signal
```

With respect to the CSP implementation the security properties SK1-SK4 may be specified as follows.

**SK1: Implicit Key Authentication (*Secrecy*).** It is assumed an Entity possesses a unique set of keys. Secrecy of a given key can then be framed as a traces property in terms of 'signalling' and 'leaking' a secret. If an honest entity A signals a secret with an honest entity B ($Claim\_Secret.A.B.key$) then the intruder must not be able to get hold of and leak the secret ($leak.key$). This characterisation is taken from [11].

```
-- leak.key can only occur before signal.Claim_Secret?p1?p2:otherthan(p1).key
SK1_SECRECY_(key) =
        let
          OPEN(key) = (signal.Claim_Secret?p1?p2:otherthan(p1)!key -> CLOSED(key))
                        [] (leak.key -> OPEN(key))
          CLOSED(key) = signal.Claim_Secret?p1?p2:otherthan(p1)!key -> CLOSED(key)
        within
          OPEN(key)

-- The property is only concerned with the keys of CORRECT entities.
SK1_SECRECY = ||| ent:CORRECT, key:inter(MyKeys(ent),Secrets) @ SK1_SECRECY_(key)

alpha_SK1_SECRECY = {|signal.Claim_Secret,leak|}
```

Note that the above process allows for a leak before secrecy is claimed. This feature allows for the re-use of keys in more general CSP models. However, for the Local Authentication Service model this feature has no effect since if the leak can occur before it can also occur after a claim.

**SK2-SK3: Key Confirmation (*Agreement*) & Authentication.** Authentication is interpreted as 'Entity Authentication' and a standard approach follows. Whenever the Entity 'commits' to a protocol run, the intended local TTCB must also have been (previously) 'running' the 'same' protocol run. A protocol run may be uniquely identified by the challenge used. This only in part defines the notion of sameness, but seems sufficient given that Key Confirmation is given a separate property.

Key Confirmation is interpreted as an agreement using the same interpretation above. The key that is established is therefore included so that 'same' protocol runs are identified by the tuple (key,challenge).

```
-- The specification may be divided between 'legitimate' running-commit pairs...
AUTH1(ttcb,ent,k,c) =
        signal.Running.ttcb?p1:otherthan(ttcb)!<k,c> ->
                signal.Commit.ent.ttcb.<k,c> -> SKIP

-- ...and all possible 'running' TTCB loners.
AUTH2(ttcb,k,c) = signal.Running.ttcb?p1:otherthan(ttcb)!<k,c> ->
                        AUTH2(ttcb,k,c)

AUTHENTICATED(ttcb,ent,k) =
        (member(ent,CORRECT) & ||| c:MyChallenges(ent) @ AUTH1(ttcb,ent,k,c))
                  ||| (||| e2:ENTs,c:MyChallenges(e2) @ AUTH2(ttcb,k,c))
```

```
-- All authentications, currently considers *all* entities as a TTCB may signal
-- Running on CORRUPT keys/challenges. However, CORRUPT entities will not use
-- signals themselves. (so Commit is not necessary here).

AUTHENTICATION = ||| ttcb:TTCBs,ent:ENTs,k:MyKeys(ent) @
                        AUTHENTICATED(ttcb,ent,k)

alpha_AUTHENTICATION = {|signal.Running,signal.Commit|}
```

Note that for a given protocol run, the authentication specification is decomposed into correct authentications (AUTH1) and incorrect ones (AUTH2).

**SK4: Trusted Against Known-Key Attacks.** For this property, the Spy is given a set of keys (and challenges) that may have been previously compromised. This specification is a static model in which a predefined compromise may be simulated by the Spy but not directly induced in an Entity. It is argued that all correct protocol runs are independent (messages uniquely identified), and so any compromised message should not be accepted by a correct Entity.

### 4.3.4 Running the Scripts

The CSP$_M$ scripts have been organised into a modular, hierarchical structure of files. Dependencies are handled automatically by a Make file which takes a stand-alone .cspm source file and creates a .csp script including the respective source file and all others it depends upon. Inclusion is implemented by the CSP$_M$ 'include' mechanism and dependents must be explicitly named by a 'depend' file.

There are five basic library scripts that this analysis and all others depend on. These include utils.cspm, types.cspm, user.cspm, time.cspm and dataexchange.cspm. Each analysis then defines a protocol script, a properties script and a checks (FDR asserts) script, as well as other protocol-specific scripts. An illustration of the dependencies between scripts is shown in figure 4.2.

The compilable CSP$_M$ scripts for the LAS protocol are to be posted in the public section of the MAFTIA home-page at [48]. It is intended that all

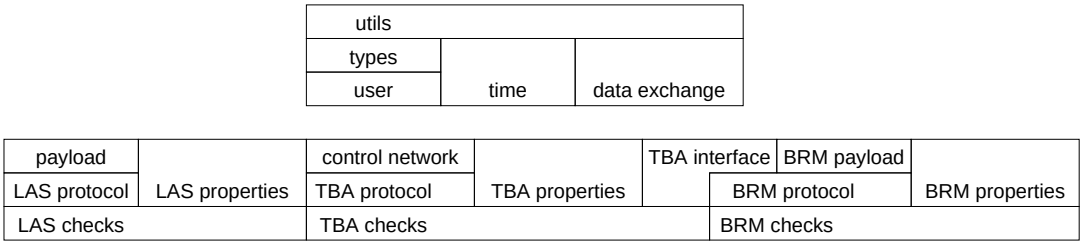


Figure 4.2: The TTCB Verification CSP$_M$ File Dependencies. All protocols depend on the library source files utils.cspm, types.cspm and user.cspm.

CSP$_M$ models developed under WP6 will eventually be posted on this site for download.

The full CSP$_M$ LAS protocol model is defined as follows:

```
alpha_AKE = Union({alpha_LOCALTTCB(t),alpha_ENTITY(e) | t:TTCBs, e:CORRECT})

PROTOCOL = ((||| ttcb:TTCBs @ LOCALTTCB(ttcb))
           |||
           (||| entity:CORRECT @ ENTITY(entity)))
                [ alpha_AKE || alpha_PAYLOAD ]
                        PAYLOAD
```

### 4.3.5 Initial Results

The initial analysis of the LAS protocol consisted of one local TTCB, one Entity and no corrupt Entities, with the aim of establishing the correctness and complexity of the CSP$_M$ protocol model. The subset $ENTs$ was set to $\{Entity\}$, $TTCBs$ to $\{TTCB\}$ and $CORRUPT$ to the empty set $\{\}$. In terms of protocol runs, the Protection parameter was restricted to a singleton set and all other parameters restricted to sets of two (Key,Challenge) and three (Eid) elements.

The number of runs an Entity may participate in is bounded by $nC$, the number of Challenges it is assigned, and the number of runs a TTCB may service is similarly bounded by $nE$, the number of Entity Ids it is assigned. The maximum number of protocol runs that may occur is given by $min(nC, nE)$. In the initial analysis this is $min(2, 3) = 2$. As described on page 87 there are currently no deductions.

Each of the component processes $LOCALTTCB$, $ENTITY$ and $PAYLOAD$ were tested with a process animator Probe [52], and checked for simple properties such as deadlock freedom and adherence to their alphabets. The checks are specified in the file LAS_checks.cspm. For example, the following refinement check establishes that the local TTCB is deadlock free (until it terminates).

```
assert SKIP [F= LOCALTTCB(TTCB) \ alpha_LOCALTTCB(TTCB)
```

In other words with all its events hidden, the process $LOCALTTCB(TTCB)$ must always behave like the process SKIP.

For the payload network, as each message cell $MEDIUM$ contributes two states and cells are independent the network has an exponential state and transition complexity (dictated by $correct\_msg$). As such, the message set must be kept small to keep the analysis tractable. With just one local TTCB, one correct Entity and the parameters above the message set contains ten messages; four possible requests and six possible replies for the two possible protocol runs.

In general, for $n$ messages the $PAYLOAD$ has exactly $2^n$ states and roughly $n.2^n$ transitions (for each super-state there are a constant number of possible transitions - from any of the $n$ message cells).

For the local TTCB and Entity, the state space is much smaller. This is especially true for the local TTCB which only processes requests in sequence. The Entity interleaves protocol runs and so the state space will clearly be larger. For the initial configuration (one local TTCB, one correct Entity, two protocol runs), the local TTCB has only 23 states and 40 transitions while the Entity has 625 states and 1584 transitions.

In terms of complexity, as the local TTCB accepts requests in order and has no memory of the requests the complexity of states is linear in each of the parameters, $nK$, $nC$, $ENTs$, and so on that determine the number of possible requests. The Entity on the other hand is much more complex. For each protocol run defined by $AKE\_ENTITY$, the number of states is linear in $nE$ the number of Entity Ids (choices) assigned to a local TTCB. However, when protocol runs can interleave and are selected from a large space of parameters the overall complexity is polynomial in $nK$, $nE$, $TTCBs$ (order determined by $nC$) and clearly exponential in $nC$, the number of challenges or rather protocol runs.

Overall, the complete protocol model $PROTOCOL$ runs to roughly 3000 states. Adding a corrupt entity increases the size of the message set to 16

and the $LOCALTTCB$ states to 77. The new protocol model then contains roughly 24,000 states. Without deductions or corrupt entities, FDR checks only a fraction of the state space of the process $PAYLOAD$. Further increases are just about manageable. With one TTCB, two correct Entities and one corrupt Entity the message set is 26 (8 encryptions and 18 signatures) and the protocol models then contains over two million states.

There are known techniques to scale down or manage the state space explosion. However, for lack of time and experience these techniques have not been explored.

### 4.3.6 Results

The security properties SK1-SK4, transcribed into the CSP$_M$ specifications SECRECY (SK1) and AUTHENTICATION (SK2-3) were verified (in the traces model) against the above protocol model configuration (one local TTCB, two correct Entities, one corrupt Entity, $nK = 2, nC = 2, nE = 3$). As expected, the analysis did not uncover any attacks on the Secrecy or Authentication. With no deductions, the Spy has no hope of breaching Secrecy and decrypting the initial request which (should) be the only message to contain the shared key. This is essentially the proof of correctness given by the TTCB Security Service design [53]. For the Authentication, the Spy has a better chance since he only has to introduce confusion.

To illustrate the properties of the CSP$_M$ protocol model, sanity checks may be performed. For instance, checks may be used to show the feasibility of certain scenarios or the protocol model may be injected with faults to test the verification process. The checks for defined scenarios are listed in the file LAS_checks.cspm, the injection of faults is a manual process.

The following example illustrates a scenario in which Entities reuse challenges. Under the current model this should not impact on Secrecy but Authentication is clearly at threat. To introduce the fault the control process $ENTITY\_$ is ammended so that it does not remove used challenges from its list of available challenges. The two CSP$_M$ specification checks are then re-run. The Secrecy check passes but the Authentication check fails,

providing the following attack trace:

$$\begin{aligned}
&\langle input.Entity.local_auth.TTCB.Ka.p.Ne,\\
&send.Entity.TTCB.E.Ku.T.\langle Ka, p, Ne\rangle,\\
&receive.TTCB.Entity.E.Ku.T.\langle Ka, p, Ne\rangle,\\
&input.Entity.local_auth.TTCB.Kb.p.Ne,\\
&\mathbf{signal.Running.TTCB.Entity}.\langle \mathbf{Ka}, \mathbf{Ne}\rangle,\\
&send.TTCB.Entity.C.S.TTCB.T.\langle Ne\rangle.T.\langle eid\rangle,\\
&send.Entity.TTCB.E.Ku.T.\langle Kb, p, Ne\rangle,\\
&receive.Entity.TTCB.C.S.TTCB.T.\langle Ne\rangle.T.\langle eid\rangle,\\
&signal.Claim_Secret.Entity.TTCB.Kb,\\
&\mathbf{signal.Commit.Entity.TTCB}.\langle \mathbf{Kb}, \mathbf{Ne}\rangle\rangle.
\end{aligned}$$

Here the attack uses two runs, between *Entity* and local TTCB $TTCB$. All protocol parameters (Key, Challenge, etc) have been renamed and the events signal events used by the Authentication set in bold. The attack is very simple; the reply from the first request is taken to be the reply from the second (which has not yet been received by the TTCB).

Although the analysis did not reveal any attacks on the security properties, it did raise an issue with the protection of the Entity Id returned by the local TTCB to an Entity. Clearly, the Spy may replace the Entity Id with any value he chooses without detection by the Entity. The effect this has depends on the future use of the Entity Id, key pair established. In the short term it creates a mismatch, in the long term it is unlikely to cause a serious breach of security as the mapping of Entity Ids to Entities is generally assumed public knowledge. For instance, it is an assumption of the TBA protocol that all Entities agree *a-priori* over the Entity List partly defining the Agreement run (see page 108).

This potential issue was raised by email with the authors [49] of the TTCB security services, acknowledged as worth changing and a possible ammendment given. In the ammended version of the protocol the Entity Id is protected using the shared key. This version has not been analysed but may well introduce new issues as the shared key is now used within the protocol. For instance, the Spy may replay the request to the TTCB any number of times, receiving a fresh encryption (same key) each time.

Alternatively the Entity Id may be protected in a similar manner to the signed challenge, using the local TTCB's private key. This seems more

natural.

### 4.3.7 Improvements

The current modelling of the Local Authentication Service could be improved in a number of areas including completeness, simplifications and faithfulness.

The analysis so far has not considered any proof techniques such as data-independence. These could be used to decompose the analysis into more tractable pieces and also to extend the (finite) checks to general (infinite) results. This would be much more satisfying than the current status in which the security properties have only been verified to a limited number of participants with limited behaviour (finite sets of keys, challenges and so on).

There a few areas where the models could be simplified. As noted in section 4.3.2, the Entities and local TTCBs currently always accept junk messages from the payload network regardless of their progress through the protocol. The messages are unnecessary and do not affect the behaviour of the Spy, Entities or local TTCBs. Certain parts of the protocol may also be safely simplified, including the protection parameter and the Entity Id. There are guide lines for such simplifying transformation detailed in chapter 8 of [11].

The current analysis has not used any deductions on the part of the Spy. Any reasonable one-message deductions are hard to justify for the Local Authentication Service given the standard 'unfaithful' cryptographic assumptions outlined in section 4.1.1. If the assumptions are weakened and more esoteric properties encoded into the deductions then perhaps new issues might be revealed.[2]

Lastly, the verification process is a very manual process with many opportunities for human error. While the interpretation and transcription of security properties and the design of the network and Spy are activities that demand special attention and consideration, the modelling of the actual protocol principals is often straight forward and amenable to automation. Most protocol descriptions involve a defined series of messages between legitimate

[2] Especially if the ammended version of the LAS protocol uses the shared key in some manner.

principals using standard manipulations suggesting the use of a tool to auto-generate the $\text{CSP}_M$ processes and channel definitions as required.

## *4.4 The Trusted Block Agreement Service*

The Trusted Block Agreement Service, in comparison to the Local Authentication Service, presents a broader challenge to model checkers such as FDR. Although there is less potential for malicious activity (the service operates over a trusted network using trusted nodes) the protocol implementing the service is much more detailed[3]. In particular, to verify the timeliness property AS5 the components that support it must incorporate a notion of time. Lastly, the protocol describes the manipulation of data tables. While perhaps feasible to model, it is more satisfying to abstract away from these details as will become apparent.

The TBA protocol has been modelled with respect to the local TTCB interface. As such, the protocol is composed of a number of local TTCBs nodes ($TBA\_NODE$), the control network ($CONTROL\_NETWORK$) and timing processes ($TIMING$). Currently, the control network process is not used and nodes communicate directly with each other though a broadcast primitive.

### 4.4.1 Design Decisions

This section describes the design decisions taken during the modelling of the TBA protocol. The full TBA protocol executed by each local TTCB is given by figure 4.3. The *broadcast* and *receive* routines are time-triggered while the *propose* and *decide* routines are called asynchronously and form the user interface to a local TTCB.

While the TBA service is presented to untrusted users over an untrusted network (the payload network), communications between the users (Entities) and the service provider (the local TTCB) are assumed to be protected (Authenticated, Integral) and clearly do not (and should not) carry secret information. As the service properties AS1-AS5 do not depend on the payload network or Entities the protocol will be modelled with respect to the

[3]but not necessarily more complex.

```
propose routine
1   when entity calls TTCB_propose(eid,elist,tstart,decision,value) do
2       if (entity already proposed) or (eid ∉ elist) or (clock() > tstart) then return error;
3       insert (elist,tstart,decision,eid,value) in sendTable;
4       get R ∈ dataTable : R.elist = elist ∧ R.tstart = tstart ∧ R.decision = decision;
5       if (R =⊥) then R := (get_tag(), elist, tstart, decision, ⊥); insert R in dataTable;
6       return R.tag
broadcast routine
7   when clock() = round_s × T_s do
8       repeat Od + 1 times do broadcast(sendTable);
9       sendTable := ⊥; round_s := round_s + 1;
receive routine
10  when clock() = round_r × T_r do
11      while (read(M) ≠ error) do
12          foreach (elist, tstart, decision, eid, value) ∈ M.sendTable do
13              get R ∈ dataTable : R.elist = elist ∧ R.tstart = tstart ∧ R.decision = decision;
14              if (R =⊥) then R := (get_tag(), elist, tstart, decision, ⊥); insert R in dataTable;
15              insert value in R.vtable;
16      round_r := round_r + 1;
decide routine
17  when entity calls TTCB_decide(tag) do
18      get R ∈ dataTable : R.tag = tag;
19      if (R ≠⊥) and [(clock() > R.tstart + T_agreement) or (all entities proposed a value)] then
20          return (calculate result using function R.decision and values in R.vtable);
21      else return error;
```

Figure 4.3: Trusted Agreement Service internal protocol. Instance at a local TTCB.

local TTCB interface, i.e. Entities input and output directly to their local TTCB. Correspondingly, the service properties AS1-AS5 will be interpreted in terms of the ability of a local TTCB to release a decision, as all Entities have equal access to their local TTCB user interface.

The timeliness property AS5 depends on the timeliness of the local TTCBs and control network. With respect to timing: (1) each local TTCB is synchronised to a global clock with precision $\pi$; (2) the protocol code is executed in real-time (and have worst case execution times); (3) the control network provides a broadcast primitive with a bounds on message latencies and omissions. The property itself specifies a maximum period of time $T_{agreement}$ taken for the last proposal to propagate to all local TTCBs. In

particular, the time constant $T_{agreement}$ is specified by [49] in terms of the maximum message latency $T_{send}$ and timings of the local TTCBs. A more detailed verification may therefore be achieved by modelling these parameters explicitly. The verification will use a standard discrete model of time using a coarse, discrete granularity (i.e. 'tock' events).

The service properties AS1-AS4 are all specified in terms of an agreed decision, the result of each local TTCB applying a pre-determined decision function to the set of proposals it has received. To verify all possible decision functions, each property will be generalised to range over sets of proposals instead of the application of a given decision function.

The user interface to the TBA service consists of two functions, *propose* and *decide*. Each function returns error codes when certain conditions are not met, for instance Entities may not propose twice or after the given 'propose by' time *tstart*. The CSP modelling of the user interface shall instead refuse requests when the conditions are not met. While this simplification is potentially dangerous, it can be assumed that handling erroneous requests does affect the local TTCBs state or timing. It can also be argued that the occurrence of an erroneous request does not give the (external) observer any new information (such as whether an Entity has proposed already).

For a given local TTCB, the TBA protocol uses two data tables, a *sendTable* and a *dataTable* (see figure 4.3). Each data table effectively buffers proposals; the *sendTable* buffers the proposals received locally through the propose function to a broadcast routine while the *dataTable* collects all proposals for a given agreement, to later process using the decision function. As such, the data tables will be modelled as BUFFER processes between the various routines. It is assumed that read-write conflicts are guarded against in any implementation.

In the TBA protocol, local TTCB nodes communicate with each other (including themselves) at a regular, synchronised period $T_s$ known as the broadcast period. Nodes use an broadcast primitive *broadcast* that is unreliable but satisfies the abstract network properties pertaining to message latencies and omissions. Each node broadcasts the set of proposals it receives locally, so that eventually all nodes hold the same set of proposals for a given agreement. All nodes must broadcast even if they have no proposals to broadcast.

Local TTCB nodes overcome the unreliability (omissions) of the broad-

cast primitive by repeating their broadcasts a number of times (in succession) and allowing enough time to elapse to guarantee arrival. The algorithm of figure 4.3 is therefore parameterised by the (known) omission degree $Od$, so that nodes broadcast $Od + 1$ copies of every message to guarantee delivery of at least one copy. All messages received by $read(M)$ must therefore be filtered for duplicates.

In modelling the communications we have chosen to abstract away from the reliability of the broadcast for the time being on the assumption that the above mechanism ($Od + 1$ repetitions) may be safely replaced by a reliable broadcast primitive. Under this abstracted model a local TTCB node simply broadcasts once and expects the message to arrive before a longer latency, $(Od + 1) \times T_{send}$, where $T_{send}$ is the original maximum message latency. The role of the control network is reduced under this model and in the current protocol model it is excluded (nodes communicate directly with each other). In addition, nodes always broadcast in a fixed order during the broadcast period. This simplification should allow a naive modelling of the broadcast routine in which arbitrary sets of proposals are broadcast.

Lastly, the assignment of Entities (Entity Ids) to local TTCBs is currently fixed. Each local TTCB has a finite set of Entity Ids it will accept, the same set as used in the Local Authentication Service and defined by the function $MyEids$ (see section 4.2).

### 4.4.2 The local TTCB Node

The local TTCB nodes is modelled by the process $TBA\_NODE$, composed of processes representing the four routines of the TBA protocol from figure 4.3. A diagram of the structure of the local TTCB node is given by figure 4.4.

In addition to the basic MAFTIA datatypes and TTCB datatypes defined in section 4.2 the TBA protocol introduces a few more functions and types mainly for convenience. In particular, name types are introduced to define an agreement and a proposal. An agreement is uniquely identified by the compound Entity List ($Elist$), 'propose by' time ($TStart$), and the (predetermined) decision function chosen ($Decision$). A proposal is then defined with an agreement ($Agreement$), an Entity Id ($Eid$) and a value proposed ($Value$).

```
-- An 'Agreement' is identified by the compound Elist.TStart.Decision
nametype Agreement = Elist.TStart.Decision
```

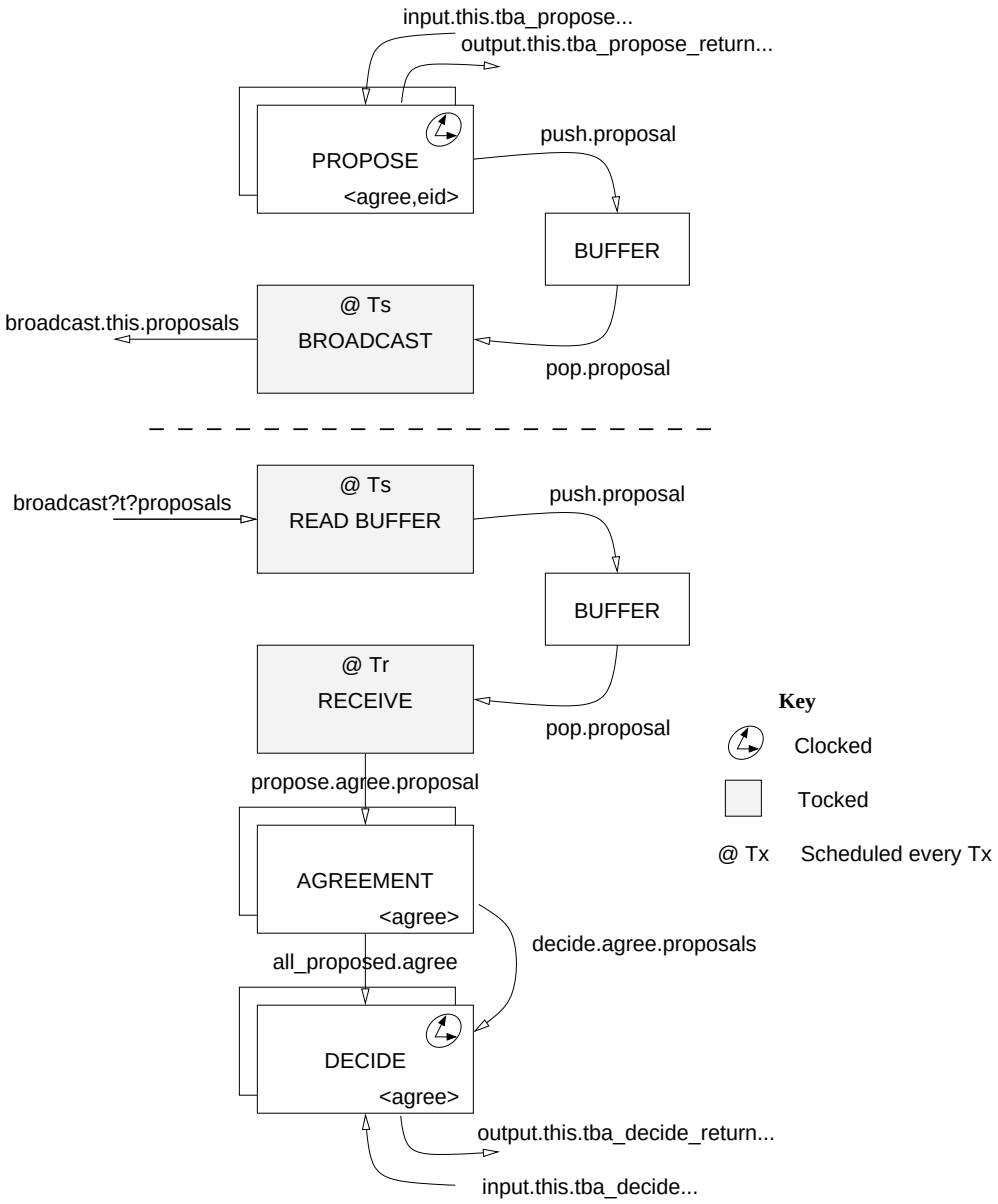


Figure 4.4: The $\text{CSP}_M$ local TTCB Node.

```
-- For each agreement, there are a finite set of proposals.
nametype Proposal = Agreement.Eid.Value
```

Although the control network is not modelled explicitly, a message set $msg$ is still defined in the usual manner to represent the set of all possible messages. In the current model, nodes synchronise on an atomic *broadcast* event to represent the simultaneous delivery of a message broadcast from a certain node. The message set $msg$ is built from all the messages that nodes may broadcast, defined for a given node by the function $TTCB_B(t)$. A message is defined as a set of proposals as above. As agreement protocol runs can run concurrently, a message may contain proposals for more than one agreement.

```
-- The 'M' function.
-- TTCBs send sets of valid proposals to each other.

TTCB_M(a,_) =
        let
          eid_order(elist) = filter(MyGroup(elist),MyEids(a))
          domain_seq = <elist.t.d.e | elist<-Elist_Order,t<-TStart_Order,
                                      d<-Decision_Order, e<-eid_order(elist)>
          msg_functions = partial_function(domain_seq,Value)
          proposals(f) = {elist.t.d.e.value | (elist.t.d.e,value) <- f}
          all_msg = {proposals(f) | f <- msg_functions}
        within
          all_msg

TTCB_B(a) = Union({TTCB_M(a,b) | b<-TTCBs})

M(a,b) = TTCB_M(a,b)

msg = Union({TTCB_B(t1) | t1 <- TTCBs})

channel broadcast : TTCBs.msg
```

The set of all possible messages a local TTCB may broadcast is constrained by the mapping of Entities (or rather Entity Ids) to local TTCBs and the fact that Entities only propose once for a given agreement (proposals must be functional). The resulting set grows exponentially but is just about manageable given at most two agreement runs, and a limited number of Entities and local TTCBs.

The user interface to each local TTCB is of a similar form to the LAS protocol interface. This time the local TTCB exports two functions *propose*

and *decide*. A typical proposal is given by the event:

$$input.TTCB.tba\_propose.(ElistA.5.1).1.2$$

in which an Entity with Entity Id 1 proposes the value 2 to the local TTCB, $TTCB$. The particular agreement denoted by $ElistA.5.1$ is composed of a symbolic entity list $ElistA$[4], a 'propose by' time of 5 and the decision function 1. Note that in the current model the value proposed can only be taken from the set $\{1, 2\}$, i.e. a binary decision. This assumption relates to the data-independence of the protocol model to proposed values (the type $Value$).

```
-- USER is a set of ENTs that are allowed to initiate requests
-- (use input/output), and all TTCBs.
USER = union(CORRECT,TTCBs)

-- Service Requests, i.e. the API call to the local TTCB.
datatype service_request = tba_propose.Agreement.Eid.Value |
                           tba_decide.Agreement

-- Service Replies.
datatype service_reply = tba_propose_return.Agreement |
                         tba_decide_return.Agreement.Value.Set(Eid)

-- The Entity Input-Output Channels --
channel input : USER.service_request

channel output : USER.service_reply
```

It should be noted that it is an implicit assumption of the TBA protocol that Entities agree *a-priori* over the parameters of the agreement. In other words: (1) they agree over the Entity list and their identities, (2) they agree over the start time *tstart*, and (3) they agree over the decision function. Note that malicious Entities cannot propose using incorrect Entity Ids as a local TTCB always knows the (Entity Id, shared key) pair established by the LAS protocol. Hence, use of the shared key during the TBA protocol authenticates the Entity Id. Confusion may well arise out side of the scope of the protocol, however these possibilities are not part of the current analysis.

In terms of time, all model components that contribute to the time constant $T_{agreement}$ of service property AS5 ($Timeliness$) must implement the

[4] The Entity Ids for an Entity list may be retrieved by the function, $MyGroup(elist)$.

timing specifications as given by the protocol description of [49]. The time constant $T_{agreement}$ is specified in [49] as:

$$T_{agreement} = T_s + WCET_{send} + T_{send} + T_r + WCET_{receive} + \pi$$

where $T_s$ and $T_r$ are the broadcast and receive periods, $WCET_{send}$ and $WCET_{receive}$ are the worst case execution times of the broadcast and receive routines, $T_{send}$ is the maximum (broadcast) message latency and $\pi$ is the precision of the local TTCB clocks. The equation considers proposals that arrive just in time (before *tstart*) and simply adds up all the maximum delays along the path from the proposal routine to the completion of the receive routine. Note that it is an assumption that the propose routine sends the proposal to the broadcast routine *before tstart* (the other functions such as initialising the dataTable are replicated by the receive routine).

In modelling the timing of the propose and decide routines, I have used a clock event to signal absolute time (in terms of *tock* counts). This amounts to using a flag event as each routine only synchronises on the relevant time, respectively *tstart* and $tstart + T_{agreement}$. For the broadcast and receive routines, each is scheduled at a regular period of *tock* events and each has a worst case execution time (in terms of *tock* events). A *tock* event then corresponds to maximum message latency of the 'reliable' broadcast primitive, i.e. $(Od + 1) \times T_{send}$. Currently, all timing constants are fixed to assumed values and significantly $WCET_{send}$ and $WCET_{receive}$ are set to 1 tock.

```
-- The timing constants:
-- These constants are in terms of tock events which currently
-- corresponds to Tsend * (Od+1), i.e. the maximum latency for a reliable
-- broadcast, in which Od+1 unreliable broadcasts are used.

-- The broadcast period
Ts = 2

-- The receive period
Tr = 3

-- Tagreement: the maximum propagation delay between tstart ('decide by') &
-- all local TTCBs receiving a final, late proposal.
-- Tagreement = Ts + WCETs + Tsend + Tr + WCETr
Tagreement = Ts + Tr + 1

-- Tmax: a limit on required clock times.
-- Equal to last tstart + Tagreement + a bit.
Tmax = seq_max(TStart_Order)+Tagreement+1
```

```
-- Tzero: when time begins.
Tzero = 1

-- the current scheduling collisions between Tzero & Tmax. Note that broadcast &
-- receive take one tock. collisions are important for the receive routine which
-- listens for broadcasts & processes (receives) them (within one tock).
collisions = {m*Ts | m<-{Tzero..Tmax/Ts},
                        n<-{Tzero..Tmax/Tr}, abs(m*Ts - n*Tr) < 1}
```

The model of the local TTCB node uses the following channels. The channels $broadcast\_flag$ and $receive\_flag$ signal the scheduling and are external events, all other channels are internal to a node (see figure 4.4).

```
-- Broadcast & Receive round flags.

channel broadcast_flag, receive_flag

-- Proposal buffers.

channel push,pop : AllProposals
channel empty,full

alpha_buffer = {|push,pop,empty,full|}

-- Agreement events.

channel propose : AllProposals
channel all_proposed : Agreement
channel decide : Agreement.EVPairs

alpha_propose = {|propose|}
alpha_agree(agree) = {|all_proposed.agree,decide.agree|}
```

The clock events and scheduling flags defined above are produced by the timing processes $CLOCK$ and $TIMER$ (see time.cspm). The process $CLOCK(t1, t2)$ counts *tock* events and produces a single *clock.t* event in between every *tock* event. After $t2 - t1$ *tock* events the process terminates. The process $TIMER(n, flag, t1)$ counts to $n-t1$ *tock* events before signalling the flag event, after which it behaves as $TIMER(n, flag, 0)$. Although the $CLOCK$ process produces all possible *clock.t* events, the propose and decide routines only synchronise on the $tstart$ and $tstart + T_{agreement}$ clock events.

```
-- The two TIMERS
TBA_TIMERS = TIMER(Ts,broadcast_flag,Tzero) [|{tock}|]
                TIMER(Tr,receive_flag,Tzero)
```

```
-- The TBA clock.
-- Communicates a clock.t event in between each tock event, after Tmax SKIPs.
TBA_CLOCK = CLOCK(Tzero,Tmax)

-- The combined timing.
TIMING = TBA_CLOCK [|{tock}|] TBA_TIMERS

alpha_TIMING = union(myclocks({Tzero..Tmax}),{tock,broadcast_flag,receive_flag})
```

The propose routine is given as follows. The main body of the routine is defined by the process $PROPOSE$, the guards (time, once-only) encorporated by the process $PROPOSAL$ and all proposals for all agreements finally combined in the process $PROPOSE\_ROUTINE$. The function $at(t, P)$ (defined in time.cspm) waits for the event $clock.t$ and then behaves as $P$. Note that under this model a proposal must terminate *before* the 'propose by' time given by the event $clock.t$. The only action of the propose routine is to *push* the proposal $agree.eid.value$ onto a broadcast buffer connected to the broadcast routine.

```
-- Each local proposal is pushed onto a proposal buffer, BROADCAST_BUFFER, for the
-- broadcast to process.
PROPOSE(this,agree,eid) =
        input.this.tba_propose.agree.eid?value ->
        push!agree!eid!value ->
        output.this.tba_propose_return!agree ->
        SKIP

-- A proposal may occur once any time before tstart. At tstart the proposal
-- is withdrawn. Each PROPOSAL must synchronise on clock.tstart only. A proposal
-- is assumed to finish before its clock.tstart.
PROPOSAL(this,tstart,agree,eid) =
        let
          PROPOSAL1 = PROPOSE(this,agree,eid) ; at(tstart,SKIP)
        within
                PROPOSAL1 [] at(tstart,SKIP)

-- All proposals are considered separately, but synchronised on their
-- clock.tstart's.
PROPOSE_ROUTINE(this) = ||| tstart:TStart @
                        [|{clock.tstart}|] elist:Elist, eid:MyGroupEids(this,elist),
                                dec:Decision @
                                        PROPOSAL(this,tstart,(elist.tstart.dec),eid)

timing_PROPOSE = {clock.t | t<-TStart}
```

The broadcast routine is given as follows. The main body of the routine is defined by the process $BROADCAST$. The timing spec-

ifications (i.e. WCETs and scheduling) are included by the process $BROADCAST\_ROUTINE$. The broadcast routine takes all the available proposals from the broadcast buffer (until *empty*) and then broadcasts[5] the set of proposals before terminating. The channel *broadcast* represents the abstracted 'reliable' broadcast.

```
-- The broadcast buffer, BUFFER1, is a normal n-place buffer.
-- A given node will only broadcast its entity id proposals, MyPrposals(this).
-- In actual fact, there are at most MyNumProposals(this) during an agreement.
BROADCAST_BUFFER(this) = BUFFER1(MyNumProposals(this),push,pop,empty,full)

alpha_BROADCAST_BUFFER(this) = {|push,pop,empty,full|}

BROADCAST(this,n,proposals) =
        let
          assertion = member(set(proposals),TTCB_B(this))
        within
        empty -> (assertion & broadcast.this!set(proposals) -> SKIP
                  [] not assertion & assert_fail -> STOP)
        []
        n > 0 & (pop?p:MyProposals(this) -> BROADCAST(this,n-1,<p>^proposals))

-- The TBA broadcast routine

BROADCAST_ROUTINE(this) =
        tock -> BROADCAST_ROUTINE(this)
        [] broadcast_flag ->
                (BROADCAST(this,card(MyProposals(this)),<>);
                        tock -> BROADCAST_ROUTINE(this))
```

The receive routine is given as follows. The main body of the routine is defined by the process $RECEIVE1$, local to the scheduled and timed process $RECEIVE$. There is also a process $READER$ which reads broadcast messages for the receive routine to later process. The reader process is naturally scheduled in time with the broadcast routine ($broadcast\_flag$). A receive buffer then connects the $READER$ and $RECEIVE$ processes.

The $READER$ process implements the assumptions about the control network described in the last paragraph of section 4.4.1. In particular, most of the features of the control network have been abstracted away to an extent that nodes communicate directly with each other and the control network is not modelled explicitly. In addition, the $READER$ process ensures that nodes broadcast in a fixed order that is then preserved for the $RECEIVE$

[5] Note that an assertion is currently needed as the message set $TTCB\_B(this)$ is functional but FDR does not know that the proposals received will always be functional!

process. This measure was taken to reduce the number of interleavings and so may be removed later. Finally, normalisation has been applied to the combination of the $READER$, $RECEIVE\_BUFFER$ and $RECEIVE$ processes to resolve some of the internal complications arising from the scheduling (see section 4.4.5).

```
-- The reader listens for a broadcast from all the TTCBs in a pre-determined
-- order, T_Order. The reader assumed a reliable broadcast primitive in which the
-- sender sends up to Od+1 unreliable broadcasts and each receiver deletes the
-- duplicate messages that get through. Again, it is assumed the reliable
-- broadcast takes one tock.
-- Each proposal is fed to RECEIVE in a pre-determined order, Proposal_Order.
READER =
        let
          READER1 = ; t:T_Order @ broadcast.t?proposals:TTCB_B(t) ->
                        (; p:filter(Proposal_Order,proposals) @ push!p -> SKIP)
        within
          tock -> READER
          [] (broadcast_flag -> READER1; tock -> READER)

-- The receive buffer, BUFFER, is a normal n-place buffer.
-- Each proposal (message) is taken one at a time and simply passed onto an
-- agreement process. There are at most Num_Proposals Proposals during all
-- possible agreement runs (interleaved etc).
RECEIVE_BUFFER = BUFFER1(Num_Proposals,push,pop,empty,full)

-- For each proposal (message), the receive routine passes the proposals onto
-- an agreement process. Clearly, the routine has little functional behaviour,
-- but the timing is needed in the form of scheduling and an assumed WCET of
-- one tock.
RECEIVE =
        let
          RECEIVE1 = empty -> SKIP
                        [] pop?proposal:AllProposals -> propose!proposal ->
                                RECEIVE1
        within
          tock -> RECEIVE
          [] (receive_flag -> RECEIVE1; tock -> RECEIVE)

-- The actual receive routine consists of the reader, receive routine and receive
-- buffer. As both READER & RECEIVE are timed, they must synchronise on tock.
RECEIVE_ROUTINE = normal(((READER [|{tock}|] RECEIVE)
                                [|alpha_buffer|] RECEIVE_BUFFER) \ alpha_buffer)
```

The decide routine is given as follows. The main body of the decide routine is defined by the process $DECIDE$. The decide routine is imple-

mented in a similar manner to the propose routine except there is an extra process $AGREEMENT$ which acts as collection point for proposals. It issues two events, *all_proposed.agree* and *decide.agree.proposals*. The *all_proposed* event can only occur when all entities have proposed, while the *decide* event can occur at any time communicating the set of proposals received so far. The process $AGREEMENT$ therefore acts as a glorified buffer between the receive and decide routines.

The main body of the decide routine $DECIDE$ incorporates a signal event of the form:

$$signal.Decide.TTCB.agree.proposals$$

where *proposals* is the set of Entity Id–Value pairs received via the *decide.agree.proposals* event, for instance $\{(1,2),(2,1),(3,2),(4,2)\}$. The routine is guarded in a similar manner to the propose routine. A decision may be released (and an input accepted) as soon as all proposals have been received (*all_proposed*) or when the 'decide by' time $tstart + T_{agreement}$ has expired. There after, entities may request a decision any number of times on a first come first serve basis. Again, the decide routines for all agreements are combined in the process $DECIDE\_ROUTINE$.

```
-- An agreement process is created to buffer and remember proposals, and determine
-- when all entities have proposed a value (all_proposed). The agreement is
-- implemented as a glorified once-only N-Channel buffer, in which the pop action
-- is suppressed but replaced by a 'read all' action (decide).
AGREEMENT(agree) =
        let
          eids = MyAgreementEids(agree)
           COLLECT(P) =
                card(P) == card(eids) & all_proposed.agree -> COLLECT(P)
                [] card(P) < card(eids) &
                                propose.agree?e:diff(eids,{e|(e,v)<-P})?v ->
                                COLLECT(union({(e,v)},P))
                [] decide.agree.P -> COLLECT(P)
        within
          COLLECT({})

-- The decide routine handles the actual decision requests from entities.
-- Any entity may request a decision any number of times, without an
-- understanding of the timeliness of input events.
-- The decision is determined by the decision function and proposals received.
-- The set p_ok contains the Entity Ids of Entities that proposed the correct
-- decision value.
DECIDE(this,agree) =
```

```
        input.this.tba_decide.agree ->
        decide.agree?proposals ->
        signal.Decide.this.agree.proposals -> (
           let
            value = tba_decision(agree,proposals)
            p_ok = proposed_ok(value,proposals)
          within
            output.this.tba_decide_return.agree.value.p_ok ->
                DECIDE(this,agree))

-- A decision combines the agreement buffer, AGREEMENT. A decision, DECISION1,
-- is allowed any time after tstart+Tagreement or once all entities have proposed
-- a value (all_proposed).
-- Each DECISION must synchronise on clock.tstart+Tagreement only.
DECISION(this,tstart,agree) =
        let
          DECISION1 = DECIDE(this,agree)
          CLOCK_IT = at(tstart+Tagreement,SKIP)
        within
          normal((AGREEMENT(agree) [|alpha_agree(agree)|]
                     (at(tstart+Tagreement,DECISION1)
                         [] all_proposed.agree -> (DECISION1 ||| CLOCK_IT))
                  ) \ alpha_agree(agree))

-- All decisions are considered separately, but synchronised on their
-- clock.tstart+Tagreement's.
DECIDE_ROUTINE(this) = ||| tstart:TStart @
                         [|{clock.tstart+Tagreement}|] elist:Elist, dec:Decision @
                                 DECISION(this,tstart,elist.tstart.dec)

timing_DECIDE = {clock.t+Tagreement| t<-TStart}
```

It should be noted that although the decide routine for a given node may progress (loop) any number of times in exclusion of all other nodes, nodes become independent once they reach the decide stage. Therefore, the properties of section 4.1.2 (termination, agreement, integrity) may be checked in the traces model with the argument that nodes are always able to receive a request and signal a decision when the global clock expires.

The four routines of the local TTCB node are composed as follows to produce the whole node defined by $TBA\_NODE$. Again, normalisation is applied to the broadcast buffer.

```
-- PROPOSE_ROUTINE & BROADCAST_ROUTINE are combined over their buffer.
PROPOSE_BROADCAST(this) = normal(((PROPOSE_ROUTINE(this) |||
                                          BROADCAST_ROUTINE(this))
```

```
                                        [|alpha_buffer|]
                                       BROADCAST_BUFFER(this)) \ alpha_buffer)

-- The RECEIVE_ROUTINE talks directly with the DECIDE_ROUTINE over the
-- propose channel.
RECEIVE_DECIDE(this) = (RECEIVE_ROUTINE [|{propose.p| p<-AllProposals}|]
                                  DECIDE_ROUTINE(this)) \ alpha_propose



-- PROPOSE_BROADCAST & RECEIVE_DECIDE are synchronised over their shared timing
-- and (currently) the Nodes broadcasts (broadcast.this).

timing_BR(this) = union(inter(timing_PROPOSE,timing_DECIDE),
                                  {tock,broadcast_flag})

sync_BR(this) = union(timing_BR(this),{|broadcast.this|})

-- The complete TBA Node...
TBA_NODE(this) = (PROPOSE_BROADCAST(this) [|sync_BR(this)|] RECEIVE_DECIDE(this))
```

### 4.4.3 CSP transcription of the informal properties

As described in the introduction, the Trusted Block Agreement Service supports five service properties AS1 to AS5. With the design decisions of section 4.4.1 and the modelling of the service with respect to the local TTCB user interface, the service properties are re-interpreted as follows.

- *AS1: Termination.* Every local TTCB eventually decides on a set of proposals.
- *AS2: Integrity.* Every local TTCB decides on at most one set of proposals.
- *AS3: Agreement.* If a local TTCB decides on a set of proposals, then all local TTCBs eventually decide on the same set of proposals.
- *AS4: Validity.* Implicit. Given agreement over the set of proposals, all local TTCBs will reach the same decision using the agreed decision function.

- *AS5: Timeliness.* Implicit. The protocol model is constructed so that a local TTCB can always make a decision after $tstart + T_{agreement}$ .

To aid the CSP specification of properties, the decide routine of the local TTCB uses the *signal* channel to signal the release of a decision to an Entity. To define the new signal a new branch *Decide* is added to the *Signal* data type.

```
-- LAS: the three types of signal we need: 'secrecy', 'commitment' to the
--       other 'running'.
-- TBA: 'decide' a set of proposals to an agreement

datatype Signal = Claim_Secret.P.P.Secrets | Commit.P.P.Text | Running.P.P.Text |
                    Decide.P.Agreement.EVPairs

channel signal : Signal
```

The following auxillary processes are used in the CSP specifications below. The processes $DECIDES$ and $DECIDES2$ repeatedly signal decisions, the former allows any decision while the later only allows a particular decision ($pairs$). Finally, the process $DECIDES3$ is a process that makes a non-deterministic decision and then behaves as $DECIDES2$.

```
-- this process decides any set of proposals repeatedly
DECIDES(this,agree) = signal.Decide.this.agree?pairs -> DECIDES(this,agree)

-- this process decides a particular set of proposals repeatedly
DECIDES2(this,agree,pairs) = signal.Decide.this.agree.pairs -> DECIDES2(this,agree,pairs)

-- this process decides a set of proposals and then behaves like DECIDES
DECIDES3(this,agree) = |~| pairs:MyEVPairs(agree) @ DECIDES2(this,agree,pairs)
```

All CSP specifications use interleaving to generalise over all TBA nodes in the system for a given property. Under the traces model, with interleaving a given node may never signal and still pass the refinement check. However, all nodes should eventually be able to signal a decision once the 'propose by' time has expired or all Entities have proposed a value. This assumption is checked by replacing the process $DECIDES$ by $DECIDES3$ and checking for refinement under the stable failures model[6].

With respect to the CSP implementation the service properties AS1-AS5 may be specified as follows.

[6] All events in the implementation except the signal events are eagerly abstracted (hiding). The only 'external' events are the input/output channels

**AS1: Termination.** This property is not currently modelled correctly as it does not insist that all local TTCB nodes actually participate. The property of termination is really to do with the assumption described above, which could be checked in the failures model by the specification $ASSUMPTION$ below. This process allows nodes to signal a decision in any order. Nodes must make a decision, after which they must repeat the same decision (stronger than termination).

```
-- AS1: Termination.
-- Every local TTCB eventually decides on a set of proposals.

TBA_TERMINATION = ||| t:TTCBs, agree:Agreement @ DECIDES(t,agree)

ASSUMPTION = ||| t:TTCBs, agree:Agreement @ DECIDES3(t,agree)
```

**AS2-AS3: Integrity & Agreement.** Integrity and Agreement are naturally combined into one property per agreement run. When ever a local TTCB node signals a decision for an agreement, all other local TTCBs must signal the same decision. The process $TBA_AGREEMENT$ represents the combined property and is verified by a refinement check in the traces model.

```
-- AS2 & AS3: Integrity & Agreement.
-- Every local TTCB decides on at most one set of proposals.
-- If a local TTCB decides on a set of proposals, then all
--                          local TTCBs eventually decide on the same set of proposals.
-- Combined: If a local TTCB decides result,
--                then all local TTCBs always decide on the same set of proposals.

TBA_AGREEMENT = ||| agree:Agreement @
                        signal.Decide?t:TTCBs!agree?pairs:MyEVPairs(agree) ->
                                (||| t:TTCBs @ DECIDES2(t,agree,pairs))
```

**AS4-AS5: Validity & Timeliness.** As described above in the reinterpretation of the service properties, the properties AS4 and AS5 are made implicit in the TBA protocol model (see also section 4.1.2).

### 4.4.4 Running the Scripts

The CSP$_M$ scripts are organised as for the Local Authentication Service. The analysis defines a protocol script, a properties script and a checks (FDR asserts) script.

The compilable CSP$_M$ scripts for the TBA protocol are to be posted in the public section of the MAFTIA home-page at [48]. It is intended that all CSP$_M$ models developed under WP6 will eventually be posted on this site for download.

The full CSP$_M$ TBA protocol model is defined as follows. Note that there is no network process or malicious agent.

```
timing_NODE = Union({timing_PROPOSE,timing_DECIDE,
                        {tock,broadcast_flag,receive_flag}})
sync_NODE = union(timing_NODE,{|broadcast|})

TBA_PROTOCOL = (([|sync_NODE|] t:TTCBs @ TBA_NODE(t))
                    [|timing_NODE|] TIMING)

alpha_TBA_PROTOCOL = Union({alpha_TIMING,
                                alpha_TBA_NODE(t) | t<-TTCBs})
```

### 4.4.5 Results

The initial analysis of the TBA protocol used one Entity list containing two Entity Ids, one 'propose by' *tstart* value (5). The decision function is always set to 1 as its function is already abstracted away by considering the service properties in terms of the set of proposals a local TTCB uses. The Entity Ids for the Entity list are chosen so that they belong to different local TTCBs (i.e. two local TTCB nodes). There is then only one possible agreement run.

The initial analysis did not reveal any issues and both the properties where verified to the limited protocol run. The checks where just about manageable but slow to compile and slow to perform the actual state exploration. The compilation time of the original protocol model was improved through factoring common sub-processes and making useful model properties explicit. The state space was improved by making the message restriction stricter (Entities are only allowed to propose once), by ordering the broadcasts of local TTCB nodes (see design decisions) and by applying normalisations to parts of the local TTCB node.

In particular, normalisation of the communications between local TTCB routines proved very beneficial as the following table illustrates. The protocol model without normalisations is given by the name *control*. Normalisations that proved successful are given by the names $normal_x$, where $R$ indicates

the receive routine, $PB$ indicates the combination of the propose and broadcast routines, $D$ indicates the decide routine and *all* indicates all of the $R$, $PB$ and $D$. The exact application ($normal(P)$) is shown in the $\text{CSP}_M$ process definitions of section 4.4.2 above. Each of the entries of table 4.1 were produced by checking the protocol model for deadlock freedom using FDR.

| $Model$ | $\#TTCBs$ | $\#Entities$ | $card(msg)$ | $States\,(000's)$ |
|---|---|---|---|---|
| $control$ | 2 | 2 | 5 | 98 |
| $control$ | 2 | 3 | 11 | 1711 |
| $control$ | 2 | 4 | 17 | crashes @ 3200 |
| $normal_R$ | 2 | 2 | 5 | 35 |
| $normal_R$ | 2 | 3 | 11 | 258 |
| $normal_R$ | 2 | 4 | 17 | 3476 |
| $normal_R$ | 3 | 3 | 7 | 1523 |
| $normal_{PB}$ | 2 | 2 | 5 | 57 |
| $normal_{PB}$ | 2 | 3 | 11 | 720 |
| $normal_D$ | 2 | 2 | 5 | 63 |
| $normal_D$ | 2 | 3 | 11 | 1547 |
| $normal_{all}$ | 2 | 2 | 5 | 9 |
| $normal_{all}$ | 2 | 4 | 17 | 459 |
| $normal_{all}$ | 3 | 3 | 7 | 260 |

Table 4.1: The state space of the full TBA protocol for one agreement run and normalisations of the component routines.

Using the improved protocol model incorporating normalisations, the service properties where also verified for a system of two agreement runs using two local TTCBs and two Entities. No new issues were revealed as would be expected for a closed system if the two runs were independent of each other.

To test the TBA protocol model several sanity checks are included in the checks (FDR asserts) script. These include the standard deadlock and livelock freedom tests as well as alphabet checks and stop-checks. The timing parameters may also be tested. If the timing parameter $T_{agreement}$ is reduced then local TTCB nodes may release a decision prematurely.

The trace given in figure 4.5 shows a disagreement when two time units are taken away from $T_{agreement}$. The parameters $tstart$ and $T_{agreement}$ are set

to 2 and 4, so any proposals must arrive before the *clock*.2 event and decisions may be released after the *clock*.6 event. Note that the broadcast period $Ts$ is set to 2 and the receive period $Ts$ to 3. The *tock* and *broadcast* events have been hidden from the trace. Inspecting the internal events of local TTCB $TTCB1$ using the FDR debugger reveals that although the $receive\_flag$ event occurs, the receive routine fails to pass on the proposal $(4, 1)$ in time for the decide routine to release.

$\langle clock.1,$
$broadcast\_flag,$
$input.TTCB.tba\_propose.ElistA.2.1.4.1,$
$output.TTCB.tba\_propose\_return.ElistA.2.1,$
$clock.2,$
$clock.3,$
$receive\_flag,$
$broadcast\_flag,$
$clock.4,$
$clock.5,$
$receive\_flag,$
$clock.6,$
$input.TTCB.tba\_decide.ElistA.2.1,$
$\mathbf{signal.Decide.TTCB.ElistA.2.1}.\{(\mathbf{4}, \mathbf{1})\},$
$input.TTCB1.tba\_decide.ElistA.2.1,$
$\mathbf{signal.Decide.TTCB1.ElistA.2.1}.\{\}\rangle.$

Figure 4.5: A failure of Agreement between local TTCBs $TTCB$ and $TTCB1$ induced by a reduced $Tagreement$ time.

### 4.4.6 Improvements

The current modelling of the Trusted Block Agreement Service could be improved in a number of areas including completeness and rigor.

The verification and modelling has concentrated on the behaviour of the local TTCB nodes as given by the pseudo-code of figure 4.3. While this has given a positive result for the limited system configurations considered (number of parties, session parameters), the unreliable control network and (fail-silent) failures of local TTCB nodes are not currently catered for. The 'unreliable' broadcast primitive is abstracted to a 'reliable' broadcast primitive using the mechanism of figure 4.3 ($Od + 1$ repititions) while the failures of nodes are not modelled. These two areas are clearly candidates for future work. The 'reliable' broadcast primitive abstraction could perhaps simply be justified through further modelling and a formal argument.

The timing parameters of the TBA protocol are currently fixed to assumed values. This could be improved by using actual timing parameters (used in a given implementation) or generalising over the parameters in the modelling of the protocol.

The proof techniques described in section 4.3.7 such as data-independence or induction have not been explored fully. The modelling and analysis have made claims to the data-independence of certain data types in the protocol such as the $Value$ type that Entities decide over. However, formal arguments for data-independence are a matter for future work.

# 5 Faithfully Implementing Reactive Systems

## *5.1 Introduction*

General security models such as the ones presented in the prior deliverables D4 [1] and D8 [2] offer a rigorously defined foundation for specifying and proving the security of cryptographic protocols. These models have been used — by providing a corresponding formal security proof — to achieve high confidence in several protocols, such as the protocols presented in [54] or Chapter 2 of this deliverable. Further provably secure protocols are expected to follow. Now it is the time to implement such provably secure protocols in a way, which preserves the proven security properties.

As already argued in [2], it is by no means an easy task to yield correspondingly secure implementations of such protocols, which retain all security properties in the real world[1]. Given our limited knowledge about the real world, all models inherently have some abstractions which only approximate and idealise the reality. Furthermore, a model has to make tradeoffs between, on the one hand, being highly detailed and close to reality and, on the other hand, keeping the model's complexity manageable. The resulting idealisations leave a crucial gap between models and the real world: idealisations restrict the capabilities of an attacker and rule out certain classes of attacks in the model which can be serious sources of security flaws in the real world. In fact, most successful attacks against cryptographic systems identify and exploit weaknesses of the implementation.

The same holds for the "secure reactive system" models as introduced in [1] and [2], when implementing systems, which have been proven secure in these models.

**Outline** We start in Section 5.2 by shortly reviewing the possible approaches for closing the gap between the models and the real world. The way we have chosen in MAFTIA is to close the gap from the implementation side. In Section 5.3 we focus on each of the identified gaps between the models and the real world. After discussing its impact on real world security, we propose, where necessary, possible measures of implementing the correspond-

[1] In the following, we mean by "real world" the world as ruled by the laws of physics and experienced by human users.

ing abstraction in the real world. We conclude in Section 5.4 by proposing a secure execution environment for secure reactive systems.

## *5.2 Possible Approaches*

D8 discusses three orthogonal approaches to close the gap between the model's current abstractions and the real world.

The first solution was to extend the generality of the underlying system model to bring it (even) closer to the real world. This approach would force protocol designers to reason additionally about all real world threats known and integrated into the model at the time the proof is done. However, the resulting proofs would be certainly unmanageable by hand and, unfortunately, even impracticable using today's tool support.

The second approach is to model relevant real-world features in the standard models. As an example, one may model covert channels in the standard secure reactive system model as a special port over which the machine sends/leaks a restricted amount of information to the attacker. We decided not to follow this approach, because it would presuppose that every prover is aware of all crucial real-world attacks, which is certainly not realistic and puts too much burden on the prover. Furthermore, for many real world issues, such as covert channels, it is not clear how to model them adequately (e.g., the amount or type of information leaked over the covert channel).

Due to the severe problems and drawbacks of the previous approaches we decided to follow another approach which closes the gap from the real world, i.e., the implementation side. The basic idea is to suggest and provide a runtime environment that implements the model's abstractions and fulfils it's assumptions. The advantages of this approach is that it does not increase the complexity of the security proofs, since the model stays unchanged. Furthermore, implementors of protocols do not have to implement crucial abstractions themselves, but can rely on the runtime environment to eliminate or at least reduce real world security pitfalls. The open question is to what extent such an ideal runtime environment is practical, i.e., whether each abstraction is implementable and at what costs. In the remainder of this chapter, we propose possible implementations of the model's different abstractions and discuss their practicality.

## 5.3 Impacts on Real-World Security

In the following subsections we discuss possible implementations of those model abstractions that we have identified as being security-relevant in D8. We shortly summarise them and discuss possible real-world implementations that increase security of the overall system. For a more detailed discussion of the abstractions we refer the interested reader to D8.

### 5.3.1 Trust Model and Corruption Model

Given the intended structure of a standard cryptographic system, as defined in [1, 2] and summarised in Section 2.2.2, a system can take different possible structures.[2] In each possible structure, a machine may either be *correct, incorrect, or dynamically corruptible*. For the choice of adequate security measures, when implementing a system in the real world, it is of great importance, which machines are assumed to be always, i.e., in all possible structures, correct and which machines may be statically incorrect or dynamically corrupted. Here, we do not distinguish between dynamically corruptible and statical incorrect machines, since it is not always possible for the runtime environment to detect if a machine has been corrupted or not. Therefore, we assume these machines to be statical incorrect as well and choose the security measures accordingly strong.

In the remainder of this chapter, we use the term *"correct machine"* to denote machines that are correct in all possible structures and *"incorrect machine"* to denote all other machines.

As an example, consider an intended structure consisting of $n$ machines $\{\mathsf{M}_1, \ldots, \mathsf{M}_n\}$ and a $(n, 2)$-threshold trust model which assumes that out of $n$ machines at most 1 ever gets corrupted. Then, there are $n$ possible structures, each assuming a different of the $n$ machines to be corrupted. Therefore, we would have to treat each machine as *incorrect* and use accordingly strong protection measures when implementing this system.

[2] Which structures are possible depends on the trust model, the corruption model and the channel model.

### 5.3.2 Enforcement of Interfaces

The models restrict access to machines to be possible only via a well-defined interface, i.e., machines can only send messages to a fixed predefined set of machines. Enforcement of interfaces in real world implementations has three orthogonal characteristics:

1. Firstly, the runtime environment has to guarantee that a correct machine $\mathsf{M}_1$ can only be accessed by implementations of those machines $\mathsf{M}_i$, whose model counterparts have access to $\mathsf{M}_1$'s model counterpart. Additionally, the implementation has to ensure that, like in the models, a machine can only access the appropriate port of another machine.

2. Secondly, the runtime environment has to guarantee that the access to machines is restricted to sending messages, as in the models. Among other things, this means that machines must not be able to, *directly or indirectly*, access the internal state of other machines. With *direct access* to a machine's state we mean that another machine can read or write the internal state by corresponding commands offered by the runtime environment (processor commands or system calls), while *indirect access* is performed via shared variables and resources or is derived from timing behaviour. Indirect access is commonly denoted as *covert channels* and handled using information flow techniques. We separately discuss covert channels in Section 5.3.7.

3. Thirdly, *type enforcement* is required to ensure that an incorrect machine cannot manipulate the behaviour of a correct machine by sending a message of a wrong or unexpected data type. A common example are buffer overflow attacks as described in [55].

In the following section we summarise existing *access control mechanisms*, and evaluate their benefit in preventing direct access to machine's internal state. Section 5.3.8 discusses possible approaches to *type enforcement*.

#### *5.3.2.1 Access Control Techniques*

There is a wide range of access control techniques that prevent machines from accessing the state of another machine, each having different character-

istics:

**Separated hardware** The most effective, but also the most costly, protection mechanism is to run clusters of correct machines on physically separated hardware. Common examples are the execution of machines on different hosts, separated hardware modules [56, 57], or the use of smart-cards.

**Processor based Memory protection** A more cost effective solution would be to protect correct machines by protection mechanisms of today's CPUs. These mechanisms, e.g., virtual address spaces [58], can prevent that a process accesses memory regions (pages) of another process and provide elementary mechanisms enabling, when controlled by an appropriate operating system, that several processes can share the same hardware without being able to directly access each other.
A drawback of these techniques, compared to completely separated hardware, is that common hardware does not virtualise the underlying hardware completely. Therefore, special care has to be taken to prevent additional information flows (see Section 5.3.7).

**Language based** Instead of performing access control on demand at runtime, it is often possible to control access at compile time. For instance, if the topology of a system is known, an analysis of the source code may allow to check whether a machine $\mathsf{M}_1$ is allowed to access another machine $\mathsf{M}_2$, or not [59, 60]. Alternatively, object modules can be manipulated *after* compilation in such a way that unauthorised access of memory addresses becomes impossible [61]. The runtime environment can verify whether a machine implementation is allowed to access another machine, before it is executed. A hybrid technique are the so called *proof-carrying-codes* [62], where a proof is generated at compile time, which testifies that the object module obeys the security policy and which can be verified by the runtime environment.

**Virtualization** Instead of hardware mechanisms, it is also possible to virtualise hardware by software, so called virtual machines. Common examples are VMware[3] or Plex86[4]. The advantage of virtual machines

[3] `www.vmware.com`

[4] `savannah.nongnu.org/projects/plex86/`

is that control over the (real) hardware is not transfered to a potentially incorrect machine, thus allowing every command to be checked before it is executed. General disadvantages of virtual machines are that they require large resources and are comparatively inefficient. Additionally, it has to be ensured that the virtual machine itself is correct, which is difficult or at least expensive to guarantee, due to its complexity.

**Interpreter** A very similar solution (but on another level of abstraction) is the interpretation of machines by an interpreter using a machine specification language. From an efficiency point of view, interpreters can be advantageous compared to virtual machines, because interpreters do not emulate a complete virtual hardware, but offer a well-defined set of high-level commands, which can be efficiently implemented and directly executed on the processor. Additionally, interpretation of machines may allow an easy translation from machines in the model to the interpreter language.

All approaches, except the use of separated hardware, also require a correct (in the sense of Section 5.3.1) operating system that controls the applied mechanism(s). To protect machines by virtual address spaces the operating system has, for example, to ensure that correct and incorrect machines do not share a memory page.

#### *5.3.2.2 Choosing the Appropriate Access Control*

Which access control measure is appropriate for interface enforcement between machines strongly depends on the trust model underlying the secure reactive system that is to be implemented. We distinguish four *types of access: correct-to-correct, correct-to-incorrect, incorrect-to-correct and incorrect-to-incorrect.*

**Incorrect-to-correct Interfaces** These interfaces are the most critical ones, since the system's security proof is based on the assumption that interfaces are being enforced and incorrect machines will try to circumvent such interfaces wherever possible. Hence, we need the highest possible level of protection and machines assumed to be correct should, ideally, be physically

separated (see *separated hardware*) from those that may fail according to the assumed trust model.

Incorrect-to-correct interfaces can be implemented as physical connections, e.g., network links (see Section 5.3.4 for further details on the implementation of communication links).

Where physical separation is not practical, memory protection techniques should be used.

**Correct-to-correct** Enforcing interfaces between correct machines is comparatively easy, because correct machines do not try to circumvent interfaces and to access the internal state of other machines directly. Therefore, in principle, no protection mechanisms are required to enforce interfaces between correct machines. As a consequence thereof, all correct machines running on the same host may be linked into one address space to improve the implementation's efficiency and to save system resources. However, memory protection mechanisms are still useful to improve the implementation's overall robustness against faults of one machine.

Note, that the runtime environment still has to guarantee that no *external access* from incorrect machines can occur (see above).

**Correct-to-incorrect Interfaces** For the same reasons as mentioned above, correct machines do not have to be controlled (e.g., by a reference monitor), even if they access incorrect machines (nevertheless, a protection mechanism may often be required to prevent that an incorrect machine accesses correct one).

**Incorrect-to-incorrect Interfaces** Recall that all incorrect machines are modelled as one adversary machine (static corruption) or being completely under the attacker's control (dynamic corruption). Thus, from a pure security point of view, no access control is necessary for incorrect-to-incorrect interfaces, since the overall system has been proven secure under the model assumption of full adversary access to those machines. However, we advocate to use at least memory-protection or language-based techniques for these interfaces, because it increases the overall system robustness with marginal overhead.

### 5.3.3 Communication

The secure reactive system models allow machines to send and receive *arbitrarily long* messages. When receiving a message, it is implicitly clipped to a machine-dependent maximum size, such that the machine is completely independent of the clipped part of the message. Furthermore, the message transfer is *atomic*, i.e., messages are received as a whole and a message transfer *takes no time*.

#### *5.3.3.1 Message Clipping*

Obviously, arbitrarily long messages, as provided by the secure reactive systems model, cannot be handled by real-world machines, because of storage limitations. To prevent attacks like buffer overflows, the runtime environment should enforce a maximum allowed message length and clip messages that are too long. If the maximum length is hardcoded, e.g., defined by the underlying communication protocol or the operating system, the clipped message can be forwarded to the resulting machine before later fragments are received. Special care must be taken if the maximum allowed message length dynamically depends on the receiving machine's internal state. To prevent timing channels in this case, the runtime environment should not deliver the message to the destination machine *before* the last fragment of the message has been received.

#### *5.3.3.2 Atomic Communication*

In today's implementations of communication protocols (e.g., TCP/IP) it is common to fragment long messages into smaller messages with a fixed length. This can become a security concern if an implementation of a receiving machine starts its computation and sends output fragments *before* the last fragment of the input message was received from the sending machine. Imagine, for example, an implementation of a stream cipher that outputs encrypted fragments before the last plain fragment was received. This allows an adversary to change later fragments depending on the output of the prior fragments and may allow new adaptive real-world attacks against the stream cipher machine, which are impossible in the models.

To prevent such kind of misbehaviour, three different countermeasures are possible:

1. The sending machine prevents pre-calculation by encrypting all fragments and transmitting the decryption key with the last fragment.

2. The receiving machine starts its transaction only after the last fragment was received.

3. The receiving machine outputs outgoing fragments only after the last fragment was received.

As we cannot assume anything about incorrect machines, especially not that they use encryption to prevent destination machines from starting their computation before the whole message has been received, the first approach only makes sense for correct sender machines. The second and third measures aim at protecting correct machines from potential real world attacks as described above.

Comparing the two latter measures, we see that the third solution has the advantage of being more efficient, because machines can start their computation before the last message fragment has been received. However, this behaviour may create a timing channel if the adversary can observe that the machine is active (see Section 5.3.7.1).

### 5.3.4 Implementing Links

The system model uses a fixed connection structure between machines, where each link must be specified and it is assumed that machines can interact only via existing links. Thus, the implementation of the models has to ensure that machines can only communicate with other machines if the model of the system contains corresponding links.

However, this assumption does not commonly hold in real world open environments with communication techniques used today. If no special measures are taken, it is possible to send arbitrary messages to arbitrary networked machines quite freely.

Therefore, in addition to the access control measures discussed in Section 5.3.2.1, the restrictions imposed by a system's connection structure must be enforced in the real world, e.g., by means of dedicated point-to-point network

connections or filtering mechanisms based on authentication information. Such filtering mechanisms may be integrated in the runtime environment or may be performed by correct machines themselves.

Intra-host communication can be controlled by the operating system without cryptographic techniques, because the operating system knows the identity of the sending machine. However, to enforce access control rules on inter-host communication, the operating system must be able to determine the sender's identity by cryptographic techniques, e.g., based on a public key infrastructure (PKI). Furthermore, the channel model assumes each connection to provide certain qualities of service. In standard cryptographic systems, we distinguish different *channel types: insecure, authentic, secure and reliable* (see Chapter 2)

**Insecure links** are modelled as being neither private nor authentic, i.e., adversaries are modelled as being able to read and modify messages which are being sent over insecure links. Furthermore, adversaries can induce new messages on a link. Implementing such links is comparably simple, since the implementor does not need to apply physical protection or cryptographic measures, such as encryption or message authentication, since the system's security does not rely on these links being private and authentic.

**Authenticated links** guarantee that the receiver of a message can rely on the sender's identifier. To provide this feature, implementations could use cryptographic functions like signatures based on a PKI.

**Secure links** guarantee the integrity and confidentiality of messages. Based on a PKI, these properties can be provided, by cryptographic encryption and signature functions.

**Reliable links** guarantee that the delivery of messages cannot be prevented by the adversary. For real world implementations reliable links have to be implemented by separated hardware (e.g., wires) to guarantee that they are completely separated and independent of other link types (e.g., to prevent Denial of Service (DoS) attacks).

Real world implementations have to guarantee that the assumptions of the channel model hold. If these properties are not guaranteed by the implementation, its security cannot be guaranteed.

### 5.3.5 Implementing Ports

As links, ports can be implemented in several ways. A solution that is close to the model, but very costly, is to use separated hardware (e.g., contacts) for every port. If, more realistically, ports are implemented on the top of a general communication layer (e.g., as a socket, remote procedure call (RPC), or a function), special care must be taken that an incorrect machine cannot send a message to a wrong port.

### 5.3.6 Concurrency

In opposite to the models, the real world allows machines and users to be executed or act in parallel. This opens incorrect real-world machines the possibility to send messages although their model counterparts would not be able to do so, since they would not be scheduled in the model. A similar deviation from the model would be if an incorrect real-world machine would be able to send two messages to two correct machines and activate them in parallel.

Currently, we feel that there is no practical means to handle the above problems in its whole generality. Therefore, protocol designers and implementors have to consider possible real-world effects for each concrete reactive system.

### 5.3.7 Additional Information Flows

The security of implementations is only guaranteed if no information flows between machines, except those explicitly modelled by the connection structure, exist. Channels that send information using mechanisms that are not part of the model are commonly denoted as *covert channels* (Lampson defines covert channels as "[channels] not intended for information transfer at all" [63]).

Implementing general systems which prevent all covert channels is hardly achievable [64, 65, 66], especially because new channels may occur if our "model" of the real world improves. Examples of relatively late detected covert channels are Simple and Differential Power Analysis [67, 68, 69], Electro-magnetic Channels [70, 71, 67] and Fault-Injection Channels [72, 73]

(see D8 for a more detailed description).

For our purposes, it is easier, since implementations of correct machines may only accidentally use covert channels. Therefore, covert channels, which leak secret information from correct machines to incorrect ones, can be eliminated by a coordinated application of measures in the runtime environment (operating system) and implementation based measures. The following sections discuss different covert channels and measures to eliminate or at least to extenuate their impact on system security.

#### *5.3.7.1 Preventing Timing Channels*

Generally, timing channels can be prevented by ensuring that the timely behaviour of a machine does not depend on its (private) local state, which is, of course, not always possible. Another approach would be to ensure that incorrect machines do not have the possibility to measure the timely behavior of correct machines. Preventing that incorrect machine implementations can measure time at all seems to be impossible, because we can in general not prevent that the adversary can synchronise its machines. The occurring timing channels can be distinguished into three classes.

First, the execution time of a machine's operation may depend on a secret variable, e.g., a variable that is guard of a loop. This allows the adversary to deduce information about a machine's internal state by measuring the delay between sending and receiving a message. Several approaches are known for detecting this kind of covert channel in the implementations [74, 75, 76], e.g. information flow analysis of the source-code. A common solution to this problem is the use of dummy instructions where needed [76].

Second, if a machine $\mathsf{M}_1$ calls sub-machines $\mathsf{M}_{1a}$ and $\mathsf{M}_{1b}$ with different execution times, and if the decision whether $\mathsf{M}_{1a}$ or $\mathsf{M}_{1b}$ is invoked by $\mathsf{M}_1$ depends on $\mathsf{M}_1$'s internal state, adversaries may derive information about $\mathsf{M}_1$'s state from the overall execution time of $\mathsf{M}_1$ and the sub-machine. Because the implementation (and with it the execution time) of a sub-machine may change, a delay cannot be hard-coded into the invoking machine $\mathsf{M}_1$. Instead, either $\mathsf{M}_1$ or the environment has to ensure that, if the information whether $\mathsf{M}_{1a}$ or $\mathsf{M}_{1b}$ is executed is security-critical and if an adversary does not a priori know which sub-machine was invoked, the output of $\mathsf{M}_1$ is delayed independently of the sub-machine invoked. The required delay has to be set to

the maximum execution times of $\mathsf{M}_{1a}$ and $\mathsf{M}_{1b}$. If the environment prevents this class of timing channel, a delay is only required between "correct → incorrect" or "incorrect → correct" borders.

The third class of timing channel occurs if an adversary can derive whether another machine is currently executing or not. This information can be derived from a machine's I/O behaviour (e.g., blocked vs. non-blocked), or by measuring the load of the host. A countermeasure against this attack would be to use an asynchronous (non-blocking) I/O implementation. If the operating system receives and stores messages before forwarding them to the receiving machine, an adversary cannot decide whether the machine is currently blocked or not.

To prevent information flow based on the current load of a host, the operating system can assign a fixed time slot to correct machines that is also consumed if the machine is currently not active. A more efficient solution would be to assign a fixed time slot to the complete set of correct machines.

#### *5.3.7.2 Preventing Storage Channels*

Storage channels allow adversaries to indirectly deduce information about a correct machine's state via some shared variable or resource. A prominent example of such a variable is the write lock of a file.

Detecting shared variables between incorrect and correct machines is a difficult task that can not be performed automatically in general [77, 78, 79]. Basically two different approaches can be applied to prevent storage channel.

**Virtualization** Shared resources can often be virtualized by the underlying environment. From a machine's viewpoint, the variable is then never locked by another machine. A prominent example are printer spoolers, which buffer printing jobs on the hard disk and sequentially forward . A process which starts a printing job does not notice when the printer is blocked by another printing job.

However, virtualization is not that straightforward for all kinds of resources, such as files. The problem with virtualizing files is that competing write-accesses of different processes may be inconsistent and, therefore, at least one write-access cannot be correctly executed.

**Resource-management** Alternatively, shared resources can be assigned

and revoked by the underlying environment independently of the states of the machines that use the resource. Possible implementations would be to distribute resources on a first come, first serve basis (assign ressources for the whole life of a machine), or based on fixed time intervals.

### 5.3.8 Type Enforcement

To protect machine implementations from attacks like buffer or stack overflows [55] performed by an incorrect machine by sending wrong messages, the type of the message should be checked. This can be done by the machine's implementation, or, to disburden developers of machines, by the environment.

Two orthogonal measures are imaginable. First, to ensure that the machine uses correct data types for in and outgoing messages, the machine's I/O behavior should be automatically derived from interface descriptions, e.g. the formal machine specification. For example, a tool could translate a machine specification into a programming- language independent interface definition language (IDL), which can then be automatically translated into an I/O implementation by common IDL-compilers.

Second, the type of a concrete message has to be compared with the interface description, e.g., by using type-safe programming languages or by using an appropriate message parser.

## *5.4 A Secure Execution Environment for Reactive Systems*

This section synthesises our previous implementation guidelines and suggests a coherent, homogeneous development and runtime environment for the secure real world implementation of secure reactive systems.

With environment we mean the implementation of the interfaces used by the machine implementation, often denoted as TCB (Trusted Computing Base). It consists of the underlying hardware, the operating system kernel, operating system services, and additional security-critical services. To guarantee a correct execution of the environment, an adversary should not be able to control the TCB or a part of it, which has to be guaranteed by external

measures (e.g., locks or tamper-resistant hardware).

The main goal is to provide an *environment* on the top of existing hardware that allows execution of real world implementation of secure reactive systems without burdening the implementor with all aspects of the model's abstractions. The environment should bridge the gap between model abstractions and the real world as far as possible.

In the following subsections we assume that machine implementations are available as object files that can be linked together into one address space or executed as separated process.

### 5.4.1 The IT-Environment

Since the integrity of the TCB of the environment is highly critical, we assume that the IT-environment ensures that the TCB cannot be manipulated (or replaced) by an adversary. This includes the physical protection of the hardware components (e.g. a locked case) and a secure bootstrap architecture, e.g., as described in [80], to load the TCB in a secure way.

We also assume that the IT-environment is capable of providing a PKI (Public Key Infrastructure) that allows the identification of external entities and the protection of messages using cryptographic techniques, such as encryption and signature schemes.

### 5.4.2 The Operating System

The main task of an operating system kernel is to provide mechanisms that allow several applications to share the same hardware. To allow a secure execution of machine implementations, the operating system has to provide some additional *security features*.

**Protected domain** The state and code of a correct machine has to be protected against incorrect machines or hostile applications, such as viruses or worms, which are running on the same operating system. Often, operation systems provide this feature based on memory protection mechanisms of the CPU (virtual address spaces) and a memory manager that prevents that critical applications share memory pages.

A machine's state and code has also to be protected when the system

is shut down. This can be achieved by cryptographic measures that protect the integrity of machine's code (the application) and their states (the files).

**Secure Communication** To allow correct machines to enforce their own security policy, the operating system has to provide a reliable communication path, e.g., IPC (Inter-Process Communication), between local machines. Therefore, a third process must neither be able to deduce information about messages sent via IPC between two other processes, nor be able to manipulate the message or the sender id.

We also assume that local IPC is atomic, i.e., a message is either transmitted completely or not at all. In contrast to the model's general atomicity assumption, the atomicity of IPC messages is easily implementable, since IPC message have a fixed format and length.

**Reference Monitor** To enforce access control rules between protected domains the operating system has to provide some reference monitor functionality allowing the TCB to control all machines that are not assumed to be correct.

A possible system architecture that could be used to provide the security properties mentioned above is the PERSEUS[5] architecture shortly described in the following section.

#### *5.4.2.1 The PERSEUS Architecture*

PERSEUS ist a security platform for end-user devices like PDAs, Smartphones or Desktop PCs [81]. On top of the core components a small environment allows a secure execution of security critical applications and, in parallel, a conventional end-user operating system like Windows or Linux. In contrast to today's security solutions the PERSEUS platform completely controls all security-critical system ressources (memory, CPU, DMA devices, etc) and is therefore able to protect security applications from the conventional operating system.

[5] www.perseus-os.org

The PERSEUS security platform uses a small and fast microkernel as a basis. Thus, it allows a strict separation between operating system components and device drivers. This modularity keeps security critical components maintainable and therefore makes the whole system less error prone. Note that conventional operating systems are designed as one large structure without protection mechanisms between components, which makes it possible that a faulty driver can modify security components and compromise the whole system.

The PERSEUS source code is publicly available under the GNU open-source licence. In the future, it is planned to formally specify the interfaces of security critical components to allow a formal verification of their implementations [82].

### 5.4.3 Middle Layer

With the operating system features listed above, additional security services can be provided by the middle layer:

#### *5.4.3.1 Secure Installer*

A machine's implementation may be installed and removed depending on the user's wishes. The task of the secure installer is to ensure that machines are only installed/removed/replaced if the user agrees. Additionally, the secure installer has to decide, depending on the underlying trust model, whether a machine can be assumed to be correct or not. A possible trust model can be provided by the use of signed code: The user defines a list of trusted entities and their signing keys. The installer assumes machine implementations to be correct if it was signed by at least one of the trusted entities. More complex trust models are imaginable.

If the user wants to install a new machine implementation, the secure installer first checks the integrity of the object file, e.g., by verifying a suitable signature provided by the machine's vendor. If the machine cannot be assumed to be correct, it is installed in a way that allows the enforcement of access control rules.

#### *5.4.3.2 Scheduler*

The scheduler has to prevent covert channels by ensuring that incorrect machines cannot "measure" the state of another probably correct machine according to the timing behaviour of the overall system. A simple solution would be to assign fixed time slices to every process and invoke a dummy process, which consumes unused CPU resources. As all correct machines share the same process, CPU cycles are only wasted if all correct machines have finished. Another approach called lattice scheduler [83, 84] orders the execution of processes depending on their trust level. For every round, all untrusted processes are executed first, then the trusted processes and at least the components of the TCB. Only if CPU cycles are free after scheduling all TCB components, the remaining cycles of this round are consumed by scheduling a dummy process. By using this kind of scheduler, one can prevent that incorrect machine implementations can derive timing information about correct ones.

#### *5.4.3.3 Access Control*

The TCB enforces access control rules on a machine-machine and machine-port basis by a reference monitor based on the security features of the operating system (see Section 5.4.2).

As common operating systems do not provide ports on a IPC level, they have to be encoded into the IPC message by the reference monitor. Based on a concrete system specification, the reference monitor can then

1. control whether the sender is allowed to send a message to the receiver, and

2. insert the appropriate port number into the IPC message.

The decision, whether a machine is allowed to interact with another machine or not, can be derived by the reference monitor from the system model (connection structure). Additional information about the local model implementation, e.g., whether it is assumed to be correct or not, are provided by the secure installer.

#### *5.4.3.4 Inter-Host Communication*

We assume that the TCB prevents that external entities can *directly* communicate with local machine implementations to ensure that incorrect machines cannot derive the state of information correct machines or disturb their correct behaviour. Instead, external entities are only allowed to communicate with a public TCB service that guarantees the following properties:

- Identification of external machines
- Atomic communication
- Security properties of communication links

External machines are identified by means of a PKI provided by the IT-environment. To ease the implementation of machines, we propose the use of proxy machines to allow the TCB to perform cryptographic operations: if local machines communicate with external machines, the message is redirected to a so-called proxy machine (a process under control of the TCB) that signs the message using the signature key of the sending machine, encrypts it if necessary, and performs the communication protocol, e.g., the fragmentation of the message. If messages are received, the proxy processes also act as a reference monitor to enforce the access control rules.

### 5.4.4 TCPA and Palladium

Although people are debating about concerns regarding economical, social and technical consequences of TCPA (Trusted Computer Platform Alliance) and Palladium, our analysis of the TCPA Main Specification [85] has shown that those systems can be used as a basis of our *secure* system [86].

The objective of these specifications is to build a so-called *trusted computer* which mainly means a DRM (Digital Rights Management) system that prevents users from unauthorized copying of digital work[6].

Nevertheless, the provided hardware extensions can, in conjunction with a secure operating system like PERSEUS (see Section 5.4.2.1), be used to

[6]See www.cl.cam.ac.uk/ñja14/tcpa-faq.html for an overview about possible negative consequences of TCPA/Palladium).

build a secure platform that protects the rights of its users. Two important features are provided by TCPA/Palladium:

1. A physically protected hardware module that creates true random numbers. Today, random numbers are generated based on system properties and user behavior which allows an adversary with root rights to guess the 'random' values without much effort.

2. A secure bootstrap architecture. To prevent that the secure execution environemnt suggested in this section cannot be bypassed by an adversary by replacing or modifying it. Technical modifications have to be prevented or, at least, noted to the user. Commonly available hardware architectures cannot prevent the user against such kind of attacks.

## *5.5 Conclusion*

In this chapter we presented several solutions on how to bridge the gap between the models presented in D4 and D8 and the real world, which is an elementary requirement to implement them in a security-preserving way. We decided to close the gap from the implementation side to prevent that the proofs are becoming more complex (and possibly unmanageable).

Finally, a possible implementation of a secure development and execution environment for reactive systems was outlined. Based on top of conventional hardware and an existing security platform providing elementary security properties, the secure execution environment implements the model's abstractions and fulfills it's assumptions to relieve developer of machine implementations.

# 6 Final Comment

The MAFTIA project systematically investigates the tolerance paradigm for building dependable distributed systems, for security critical applications. For this it combines techniques and notions from fault tolerance and various areas of security, such as intrusion detection and cryptographic protocols. MAFTIA workpackage 6 has been concerned with the rigorous definition of the basic MAFTIA concepts, and the verification and assessment of the work on dependable middle-ware. In order to do this we have pursued two strands of research: Cryptography and Formal Modelling.

Both the cryptography and formal-methods communities are developing techniques for proofing systems secure. The former aims at proofs which rigorously deal with issues such as computational power and success probabilities of adversaries, while the latter aims at proofs that can be automatically verified of even generated. Both approaches have their limitations. Cryptographic defintions of complex systems are often sketchy and hand crafted, leaving them open to error. Current formal methods in security cannot be applied directly to cryptographic proofs, they have to make abstractions (again, open to error). In addition, the tools used to construct formal proofs, necesary for realistic sized systems, also possess limitations. In workpackage 6 we have utilised both techniques, pushing the boundaries of what can be acheived, and begun to investigate how we might best combine the approaches.

We have developed rigorous system models which allow us to split reactive systems into two layers. The lower layer is a cryptographic system whose security can be rigorously proven using standard cryptographic arguments. To the upper layer it provides an abstract service that hides all cryptographic details. This upper layer can then be verified using formal methods. The model supports composition in a manner which preserves security, and supports modular design and security proofs of complex systems. Furthermore, common types of cryptographic systems and their trust models are expressed as special cases of this model. In particular, these include systems with static or adaptive adversaries (more realistic). We have demonstrated proofs using these techniques on a variety of examples.

In addition, we have utilised the process algebra CSP and accompanying model-checker FDR to verify core MAFTIA concepts. We have developed

a library of techniques for the modelling of MAFTIA-like protocols. This library contains re-usable constructs, supporting the modular developement of proof. Novel abstractions have been created to overcome the limitations of what can be acheived using a model-checker for proof. In numerous cases the results of performing such analysis has high-lighted areas of underspecification in protocol descriptions.

Proving a system specification secure is only useful if the implementation faithfully preserves that security. Achieving this is not a trivial task, and one reason why this is so is because our specification models typically contain abstractions from the real world. In WP6 we have conducted research into how we might close the gap between the real world, and the abstraction models which we have proven secure.

We refer the reader to deliverables [1, 3, 2, 6] in addition to this, for details on the formalisation of MAFTIA concepts. Deliverables [1, 3] and this deliverable contain details of the model-checking analysis performed. Deliverables [1, 2]and this deliverable contain details of the work on rigorous system models and faithful implementations.

# 7 Appendix

The CSP processes that we use are constructed from the following CSP$_M$ constructs:

- `STOP` is the simplest CSP process; it never engages in any action, nor terminates. It is equivalent to deadlock.
- `a -> P` is the most basic program constructor. It waits to perform the event `a` and then behaves as process `P`. The same notation is used for outputs (`c!v -> P`) and inputs (`c?x -> P(x)`) of values along named channels.
- `P |~| Q` represents internal choice. It behaves as `P` or `Q` *nondeterministically.*
- `P [] Q` represents external choice. It will offer the initial actions of both `P` and `Q` to its environment at first; its subsequent behaviour is like `P` if the initial action chosen was possible only for `P`, and like `Q` if the action selected `Q`. If both `P` and `Q` have common initial actions, its subsequent behaviour is nondeterministic (like `|~|`). `STOP [] P` behaves as `P`.
- `P [| a |] Q` represents parallel composition. `P` and `Q` evolve concurrently, except that events in `a` occur only when `P` and `Q` agree to perform (i.e. *synchronise* on) them.
- `P ||| Q` represents interleaved parallel composition. `P` and `Q` evolve separately, and do not synchronise on any events. Equivalent to `P [| {| |} |] Q`
- `P [ a || b ] Q` represents alphabetised parallel. `P` and `Q` have to agree on events in the intersection of their alphabets.
- `|| x:a @ [A(x)] P(x)` represents replicated alphabetised parallel. This constructs the parallel composition of `P(x)` processes, one for each `x` in `a`, over their respective alphabets.
- `P \ A` is the CSP hiding operator. This process behaves as `P` except that events in set `A` are hidden from the environment and are solely

determined by `P`; the environment can neither observe nor influence them.

- `P [[ a <- b ]]` represents the process `P` with `a` renamed to `b`.
- `P [ a <-> b ] Q` is the linked parallel operator. `P` and `Q` synchronise on channels `a` and `b`, which have been renamed to the same and hidden. There are also straightforward generalisations of the choice operators over non-empty sets, written `|~| x:X @ P(x)` and `[] x:X @ P(x)`.

# Bibliography


[1] Formal model of basic concepts. Deliverable D04, EU Project IST-1999-11583 Malicious- and Accidental-Fault Tolerance for Internet Applications (MAFTIA), July 2001.

[2] Crypographic semantics for algebraic model. Deliverable D08, EU Project IST-1999-11583 Malicious- and Accidental-Fault Tolerance for Internet Applications (MAFTIA), February 2002.

[3] Specification and verification of selected intrusion tolerance properties using csp and fdr. Deliverable D07, EU Project IST-1999-11583 Malicious- and Accidental-Fault Tolerance for Internet Applications (MAFTIA), February 2002.

[4] *A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission*, 2001.

[5] Conceptual model and architecture. Deliverable D02, EU Project IST-1999-11583 Malicious- and Accidental-Fault Tolerance for Internet Applications (MAFTIA), November 2001.

[6] Conceptual model and architecture. Deliverable D21, EU Project IST-1999-11583 Malicious- and Accidental-Fault Tolerance for Internet Applications (MAFTIA), January 2003.

[7] Michael Backes, Christian Jacobi, and Birgit Pfitzmann. Deriving cryptographically sound implementations using composition and formally verified bisimulation. In *Formal Methods Europe '02 (FME)*, volume 2391 of *Lecture Notes in Computer Science*, pages 310–329. Springer-Verlag, Berlin Germany, 2002.

[8] C.A.R. Hoare. Communicating sequential processes. Prentice Hall (1985).

[9] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[10] Formal Systems (Europe) Limited. *FDR Failures-Divergences Refinement: User Manual and Tutorial*.

[11] Peter Ryan and Steve Schneider. *Modelling and Analysis of Security Protocols*. Addison-Wesley, 2001.

[12] First specification of apis and protocols for the maftia middleware. Deliverable D24, EU Project IST-1999-11583 Malicious- and Accidental-Fault Tolerance for Internet Applications (MAFTIA), September 2001.

[13] Specification of dependable trusted third parties. Deliverable D26, EU Project IST-1999-11583 Malicious- and Accidental-Fault Tolerance for Internet Applications (MAFTIA), January 2001.

[14] Noreddine Abghour, Yves Deswarte, Vincent Nicomette, and David Powel. Specification of authorisation services. Deliverable D27, EU Project IST-1999-11583 Malicious- and Accidental-Fault Tolerance for Internet Applications (MAFTIA), January 2001.

[15] Ran Canetti, Juan Garay, Gene Itkis, Daniele Micciancio, Moni Naor, and Benny Pinkas. Multicast security: A taxonomy and some efficient constructions. In *INFOCOMM'99*, March 1999.

[16] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[17] Ingemar Ingemarsson, Donald T. Tang, and C. K. Wong. A conference key distribution system. *IEEE Transactions on Information Theory*, 28(5):714–720, September 1982.

[18] Michael Steiner, Gene Tsudik, and Michael Waidner. Key agreement in dynamic peer groups. *IEEE Transactions on Parallel and Distributed Systems*, 11(8):769–780, August 2000.

[19] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *Advances in Cryptology – CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1994.

[20] Simon Blake-Wilson and Alfred Menezes. Entity authentication and authenticated key transport protocols emplyoing asymmetric techniques.

In Bruce Christianson, Bruno Crispo, Mark Lomas, and Michael Roe, editors, *Security Protocols—5th International Workshop*, volume 1361 of *Lecture Notes in Computer Science*, pages 137–158, Cambridge, United Kingdom, April 1998. Springer-Verlag, Berlin Germany.

[21] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. A modular approach to the design and analysis of authentication and key exchange protocols. In *30th Annual Symposium on Theory Of Computing (STOC)*, Dallas, TX, USA, May 1998. ACM Press.

[22] Victor Shoup. On formal models for secure key exchange. Research Report RZ 3120 (#93166), IBM Research, April 1999. A revised version 4, dated November 15, 1999, is available from `http://www.shoup.net/papers/`.

[23] D. Steer, L. Strawczynski, W. Diffie, and M. Wiener. A secure audio teleconference system. In Shafi Goldwasser, editor, *Advances in Cryptology – CRYPTO '88*, number 403 in Lecture Notes in Computer Science, pages 520–528, Santa Barbara, CA, USA, 1990. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany.

[24] Mike Burmester and Yvo Desmedt. A secure and efficient conference key distribution system. In A. De Santis, editor, *Advances in Cryptology – EUROCRYPT '94*, number 950 in Lecture Notes in Computer Science, pages 275–286. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1995. Final version of proceedings.

[25] Klaus Becker and Uta Wille. Communication complexity of group key distribution. In *5th ACM Conference on Computer and Communications Security*, pages 1–6, San Francisco, California, November 1998. ACM Press.

[26] Alain Mayer and Moti Yung. Secure protocol transformation via "expansion": From two-party to multi-party. In *6th ACM Conference on Computer and Communications Security*, pages 83–92, Singapore, November 1999. ACM Press.

[27] Matt Moyer, Josyula R. Rao, and Pankaj Rohatgi. A survey of security issues in multicast communications. *IEEE Network*, 13(6):12–23, November/December 1999.

[28] Giuseppe Ateniese, Michael Steiner, and Gene Tsudik. New multiparty authentication services and key agreement protocols. *IEEE Journal on Selected Areas in Communications*, 18(4):628–639, April 2000.

[29] Catherine Meadows. Extending formal cryptographic protocol analysis techniques for group protocols and low-level cryptographic primitives. In *Workshop on Issues in the Theory of Security (WITS'00)*, University of Geneva, Switzerland, Jul 2000. Electronic proceedings: `http://www.dsi.unive.it/IFIPWG1_7/WITS2000/programme-new.html`.

[30] Olivier Pereira and Jean-Jacques Quisquater. Security analysis of the Cliques protocols suites. In *14th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, June 2001.

[31] Birgit Pfitzmann, Matthias Schunter, and Michael Waidner. Cryptographic security of reactive systems. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 32, 2000. Workshop on Secure Architectures and Information Flow, Royal Holloway, University of London, December 1 - 3, 1999.

[32] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[33] Emmanuel Bresson, Olivier Chevassut, David Pointcheval, and Jean-Jacques Quisquater. Provably authenticated group diffie-hellman key exchange. In Pierangela Samarati, editor, *Proceedings of the 8th ACM Conference on Computer and Communications Security*, pages 255–264, Philadelphia, PA, USA, November 2001. ACM Press.

[34] Birgit Pfitzmann and Michael Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 2001. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.

[35] Birgit Pfitzmann and Michael Waidner. Composition and integrity preservation of secure reactive systems. In Sushil Jajodia, editor, *7th ACM Conference on Computer and Communications Security*, pages 245–254, Athens, Greece, November 2000. ACM Press.

[36] Michael Steiner, Gene Tsudik, and Michael Waidner. Diffie-Hellman key distribution extended to groups. In Clifford Neuman, editor, *3rd ACM Conference on Computer and Communications Security*, pages 31–37, New Delhi, India, March 1996. ACM Press. Appeared as revised and extended journal version as [18].

[37] Andrew C. Yao. Theory and applications of trapdoor functions. In *23rd Symposium on Foundations of Computer Science (FOCS)*, pages 80–91. IEEE Computer Society Press, 1982.

[38] Stephen J. Garland and Nancy A. Lynch. Using I/O automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 285–312. Cambridge University Press, 2000.

[39] L. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[40] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *Advances in Cryptology – EUROCRYPT '97*, number 1233 in Lecture Notes in Computer Science, pages 256–266. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1997.

[41] Dan Boneh. The Decision Diffie-Hellman problem. In *Third Algorithmic Number Theory Symposium*, number 1423 in Lecture Notes in Computer Science, pages 48–63. Springer-Verlag, Berlin Germany, 1998.

[42] Ahmad-Reza Sadeghi and Michael Steiner. Assumptions related to discrete logarithms: Why subtleties make a real difference. In Birgit Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT '2001*, number 2045 in Lecture Notes in Computer Science, pages 243–260, Innsbruck, Austria, 2001. Springer-Verlag, Berlin Germany.

[43] Birgit Pfitzmann, Michael Steiner, and Michael Waidner. A formal model for multi-party group key agreement. Technical Report RZ 3383 (# 93419), IBM Research, 2002.

[44] Michael Steiner. *Secure Group Key Agreement*. Dissertation, Naturwissenschaftlich-Technische Fakultät der Universität des Saarlandes, Saarbrücken, March 2002.

[45] Markus Stadler. Publicly verifiable secret sharing. In Ueli Maurer, editor, *Advances in Cryptology – EUROCRYPT '96*, number 1070 in Lecture Notes in Computer Science, pages 190–199. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1996.

[46] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *38th Symposium on Foundations of Computer Science (FOCS)*, pages 458–467. IEEE Computer Society Press, 1997.

[47] Probabilistic symbolic model checker. http://www.cs.bham.ac.uk/ dxp/prism/.

[48] MAFTIA Consortium. *Malicious and accidental-fault tolerance for internet applications*.

[49] Miguel Correia, Paulo Veríssimo, and Nuno Ferreira Neves. The design of a COTS real-time distributed security kernel. Technical report, Faculdade de Ciências da Universidade de Lisboa, 1999.

[50] Ranko Lazic. *A Semantic Study of Data Independence with Applications to Model Checking*. PhD thesis, Oxford University, 1999.

[51] *Proving security protocols with model checkers by data independence techniques*, 1998.

[52] Formal Systems (Europe) Limited. *ProBE Process Animator: User Manual*.

[53] Miguel Correia, Paulo Veríssimo, and Nuno Ferreira Neves. The design of a COTS real-time distributed security kernel (extended version). Technical report, Faculdade de Ciências da Universidade de Lisboa, 1999.

[54] Birgit Pfitzmann, Matthias Schunter, and Michael Waidner. Provably secure certified mail. Research Report RZ 3207 (#93253), IBM Research, August 2000.

[55] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49):File 14, 1996. Appeared also on bugtraq (`http://www.securityfocus.com/templates/archive.pike?list=1&date=1996-11-08&msg=Pine.LNX.3.91.961109134601.15637b-100000@underground.org`).

[56] J.D. Tygar and Bennet Yee. Dyad: a system using physically secure coprocessors. In *Technological Strategies for Protecting Intellectual Property in the Networked Multimedia Environment*, The Journal of the Interactive Multimedia Association Intellectual Property Project, Coalition for Networked Information, pages 121–152, MIT, Program on Digital Open High-Resolution Systems, January 1994. Interactive Multimedia Association, John F. Kennedy School of Government.

[57] Bennet S. Yee. *Using Secure Coprocessors*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1994. CMU-CS-94-149.

[58] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2nd edition, 2001.

[59] Fred B. Schneider, Greg Morriset, and Robert Harper. A language-based approach to security. In Reinhard Wilhelm, editor, *Informatics – 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, Berlin Germany, 2001.

[60] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[61] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[62] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 229–243, Berkeley, CA, USA, 1996. USENIX.

[63] Butler Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[64] J. Thomas Haigh, Richard A. Kemmerer, John McHugh, and William D. Young. An experience using two covert channel analysis techniques on a real system design. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 14–24, Oakland, CA, April 1986. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.

[65] Jonathan K. Millen. Covert channel capacity. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* [87], pages 60–66.

[66] Randy Browne. Mode security: An infrastructure for covert channel suppression. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 39–55, Oakland, CA, May 1994. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.

[67] Mehdi-Laurent Akkar, Régis Bevan, Paul Dischamp, and Didier Moyart. Power analysis, what is now possible... In T. Okamoto, editor, *Advances in Cryptology – ASIACRYPT '2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 489–502, Kyoto, Japan, 2000. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany.

[68] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Introduction to differential power analysis and related attacks. `http://www.cryptography.com/dpa/technical/`.

[69] Suresh Chari, Charanjit Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1999.

[70] Markus G. Kuhn and Ross J. Anderson. Soft tempest: Hidden data transmission using electromagnetic emanations. In David Aucsmith, editor, *Information Hiding—Second International Workshop, IH'98*, vol-

ume 1525 of *Lecture Notes in Computer Science*, Portland Oregon, USA, April 1998. Springer-Verlag, Berlin Germany.

[71] Ross J. Anderson. *Security Engineering — A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, 2001.

[72] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS. In Hugo Krawczyk, editor, *Advances in Cryptology – CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1998.

[73] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, 14(2):101–119, 2001.

[74] E. S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.

[75] D. Denning and P. Denning. Certification of programs for secure information flow. In *Communications of the ACM*, pages 504–512, July 1977.

[76] Johan Agat. Transforming out timing leaks. In *27th Symposium on Principles of Programming Languages (POPL)*, pages 40–53. ACM Press, 2000.

[77] Chii-Ren Tsai, Virgil D. Gligor, and C. Sekar Chandersekaran. A formal method for the identification of covert storage channels in source code. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* [87], pages 74–84.

[78] Richard A. Kemmerer and Phillip A. Porras. Covert flow trees: A visual approach to analyzing covert storage channels. *IEEE Transactions on Software Engineering*, 17(11):1166–1185, 1991.

[79] Richard A. Kemmerer. Share resource matrix methodology: An approach to identifying storage and timing channels. *ACM Transactions on Computer Systems*, 1(3):256–277, August 1983.

[80] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A reliable bootstrap architecture. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 65–71, Oakland, CA, May 1997. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.

[81] Birgit Pfitzmann, James Riordan, Christian Stüble, Michael Waidner, and Arnd Weber. The PERSEUS system architecture. Technical Report RZ 3335 (#93381), IBM Research Division, Zurich Laboratory, April 2001.

[82] Michael Hohmuth, Hendrik Tews, and Shane G. Stephens. Applying source-code verification to a microkernel - the vfiasco project. In *Proceedings of the 10th ACM European SIGOPS Workshop*, 2002.

[83] D. Denning. A lattice model of secure information flow. In *Communications of the ACM*, pages 236–243. ACM, May 1976.

[84] Wei-Ming Hu. Lattice scheduling and covert channels. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 52–61, Oakland, CA, May 1992. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.

[85] TCPA Technical Committee. Trusted computing platform alliance (TCPA) main specification v1.1b. Technical report, TCPA Alliance, February 2002.

[86] Ahmad-Reza Sadeghi and Christian Stüble. Bridging the gap between tcpa/palladium and personal security. Technical report, Saarland University, Germany, January 2003.

[87] IEEE Computer Society, Technical Committee on Security and Privacy. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, April 1987. IEEE Computer Society Press.