Certified Security by Design for the Internet of Things

Shiu-Kai Chin, Ph.D.

Department of Electrical Engineering and Computer Science
Syracuse University, Syracuse, New York
March 2016

Abstract

Integrity and security are essential for societies dependent on interconnected people, devices, organizations, and services. As engineers and computer scientists, our role is to provide critical infrastructure that is safe and secure. By secure, we mean that *at all levels, instructions are executed if and only if they are authenticated and authorized*. This property is known as *complete mediation*. Our goal is to provide engineers and computer scientists with the means to fulfill our purpose by providing systems with credible assurances that complete mediation is satisfied. Assurances of security and integrity require logical consistency from the top level with mission statements and commander's intent, down through organizational policies, rules of engagement and protocols, and ending with the execution of individual instructions by people or devices. We use logic and computer-assisted reasoning tools to describe and verify consistency. In particular, we use an access-control logic to reason about authentication and authorization. This logic is a command, control, and communications (C3) calculus. Our C3 calculus is implemented as a conservative extension to the Cambridge Higher Order Logic (HOL) theorem prover. Within HOL, we integrate our C3 calculus with an algebraic model of cryptographic operations and secure state machines, i.e., transition systems described using structural operational semantics. Our methods for modeling transition systems are parameterized in terms of state-transition functions, output functions, authentication functions, and security context. This makes our definitions and theorems scalable to infinite-state systems and allows for specialization to particular missions, protocols, and systems. The C3 calculus, algebraic models of cryptographic operations, and secure state machines implemented in HOL combine to give us the capability we call *certified security by design* (CSBD)[1]. To illustrate the use of CSBD, we include the development of a networked thermostat starting with a high-level CONOPS and refine it down to the level of an infinite-state machine with data structures describing delegations, jurisdiction, public-key certificates, and root trust assumptions.

[1]This technical report is the basis for Chapter 1: Certified Security by Design for the Internet of Things, appearing in Cyber-Assurance for the Internet of Things, [3]

Contents

Chapter 1

Introduction

Incorporating security into the design of components used in the Internet of Things (IoT) is essential for securing the operations of the IoT and the cyber-physical infrastructure upon which society depends. The pervasiveness of the IoT and its part in critical infrastructure requires incorporating security into the design of components from the start.

There are several challenges to incorporating security into the design of IoT components from the start. These challenges include:

1. Precisely describing confidentiality and integrity policies in ways that are amenable to formal reasoning.

2. Maintaining logical consistency among confidentiality and integrity policies and implementation at all levels of abstraction, from high-level behavioral descriptions at the user level, down to implementations at the level of state machines and transition systems.

3. Providing compelling evidence of security that is quickly and easily reproducible by certifiers.

This is not the first time the electrical and computer engineering profession has faced these challenges. In fact, the IoT is compelling evidence of successfully meeting the challenges of design, accountability, consistency, and verifiability across multiple levels of abstraction. To learn and draw inspiration from the past, we need only look back to the 1970s and 1980s when the challenges of designing and implementing very large scale integrated (VLSI) circuits was encountered and overcome.

1.1 Lessons from the Microelectronics Revolution

In the 1970s, it was inconceivable that designers of algorithms and instruction-set architectures could fashion specialized integrated circuits down to the level of physical layouts. Each level of design had its collection of design detail, e.g., transistor models at the circuit design level, and minimum separation distances among metal and polysilicon features at the layout level.

The union of all design concepts spanning algorithm design down to layouts was too much for a single designer to grasp conceptually. The prospect of a single designer accounting for all design details spanning algorithm to layout design was even more daunting. The key insight that made VLSI design possible was [9]

> "... to sidestep tons of accumulated vestigial practices in system architecture, logic design, circuit design and circuit layout, and replace them with a coherent but minimalist set of methods."

Specifically, the minimalist set of methods made use of:

- parameterization i.e., specifying λ as the maximum minimum feature size in circuit layouts,
- idealized transistor behavior as switch behavior,
- consistent interpretations of voltages, transistor state, truth values,
- interpretations linking models at multiple levels, spanning layouts to transition systems, and
- computer-aided design (CAD) tools.

Computer hardware design is often called *logic design* for good reason. Propositional logic pervades all levels of abstraction in VLSI design. Transistor circuits and layouts are related to logic operators such as *negation, nand*, and *nor*. Networks of logic gates implement arithmetic logic units, multiplexers, flip-flops, and registers that are the components of datapaths. Base 2 arithmetic is used precisely because operations on binary numbers conveniently map to logic operations. Timing and control is achieved using finite-state machines. Finite-state machines are parameterized by next-state and output functions described by propositional logic formulas and implemented by combinational logic components. Instruction-set architectures are implemented by a combination of data and control paths whose operations are controlled and sequenced by finite-state machines.

The VLSI-inspired vision for securing the integrity of the Internet of Things is this: harmonize multiple levels of abstraction by using the same logic at all levels to describe behavior at each level. This enables designs at each level of abstraction to be related to behavior at other levels. This provides the means for a continuous thread of logical consistency and a foundation for formally verified assurances of security and integrity.

The aspects of security and integrity of the IoT upon which we focus revolve around answering the question, *when given a request to execute a command within a security context of policies, authorizations, and trust assumptions, should we execute the command or not?* This question, and others like it, falls squarely within the realm of access control. Access control is a central concept behind firewalls, reference monitors, security kernels, and hypervisors. What is needed is an access-control logic that describes our security and integrity concerns in much the same way that propositional logic describes functional behavior.

For pragmatic reasons, an access-control logic, and the methodologies built upon it, must integrate well with the propositional logic, models, and design methods of computer hardware designers. As is often the case, simplicity brings the benefits of wide applicability, broad utility, and durability, as illustrated by propositional logic in hardware design. The access-control logic we use in this chapter is form of propositional *modal* logic, i.e., a logic that incorporates modes (e.g., states, worlds, configurations, or possibilities) into determining the truth value of logical propositions. This is an incremental step above the propositional logic of conventional hardware design and enables us to blend access control into machine design and verification.

Before delving into the details of a particular access-control logic, we describe the objectives of what we call *certified security by design*, provide a simple motivating application as context, and state the critical requirements that must be satisfied to make certified security by design a reality.

1.2 Certified Security by Design

Certified Security by Design (CSBD) is an approach intended to design security into systems from the start and provide credible evidence that security claims are true. The goals of CSBD are:

1. **Complete mediation**—authenticating and authorizing—all commands at all levels from high-level concepts of operations down to transition systems realized as state machines in hardware, and
2. **Formal proofs of integrity and security that are easily and rapidly verified by third parties**, similar to the way VLSI (very large scale integrated) circuits are described and verified using an array of electronic design automation (EDA) tools.

1.2.1 Concepts of Operations

Users of systems, where systems are machines, software applications, protocols, or processes coordinating the work among human organizations, typically have behavioral models of the systems they use. These models are *concepts of operations* or CONOPS. As defined by IEEE Standard 1362 [1], a CONOPS expresses the "characteristics for a proposed system from a user's perspective. A CONOPS also describes the user organization, mission, and objectives from an integrated systems point of view."

The US military has a similar definition of CONOPS in Joint Publication 5-0, Joint Operational Planning [2]. For military leaders planning a mission, a CONOPS describes "how the actions of components and organizations are integrated, synchronized, and phased to accomplish the mission."

Put more plainly, a CONOPS describes the who, what, when, and why. When we explicitly address security and integrity concerns, we state how we know with whom we are dealing and what authority they have, i.e., how we authenticate and authorize people, processes, statements, and commands.

Figure 1.1 Flow of Command and Control (C2) for a Simple CONOPS

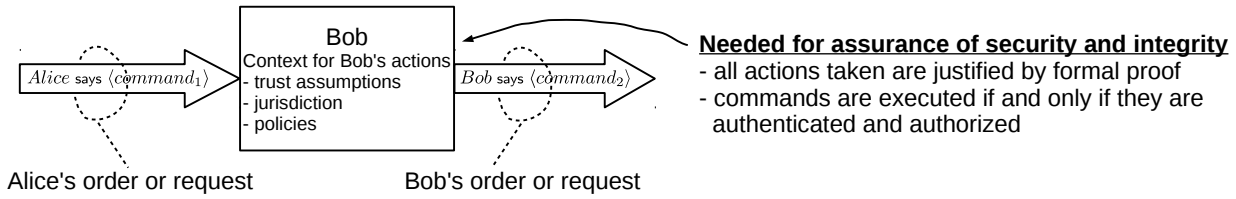


Figure 1.1 shows a diagram of a simple CONOPS. Here is its interpretation.

1. The flow of command and control in this figure is from left to right. Alice issues a command by some means (speaking, writing, electronically, telepathy, etc.). This is symbolized by

$$Alice \text{ says } \langle command_1 \rangle.$$

2. The box in the center labeled **Bob** shows Bob receiving Alice's command on the left. Inside the box are the things Bob "knows", i.e., the context within which he attempts to justify acting on Alice's command. The context might include a policy that if Bob receives a particular command, such as *go*, then he is to issue another command, such as *launch*. Typically, before Bob acts on Alice's command, his operational context includes statements or assumptions such as Alice has the authority, jurisdiction, or is to believed on matters related to the command she has made.

3. The arrow coming from the right hand side of the box shows Bob's statement or command, which is symbolized by

$$Bob \text{ says } \langle command_2 \rangle.$$

4. What Figure 1.1 shows is one C2 sequence starting from left to right. Bob gets an order from Alice. Bob decides based on Alice's order and what he knows (the statements inside the box), that it is a good idea to issue $command_2$. This is symbolized by

$$Bob \text{ says } \langle command_2 \rangle.$$

Regarding the comment in Figure 1.1, for assurance what we want is a logical justification of the actions Bob takes given the order he receives and the context within which he is operating. For us, logical justifications are *proofs in mathematical logic*.

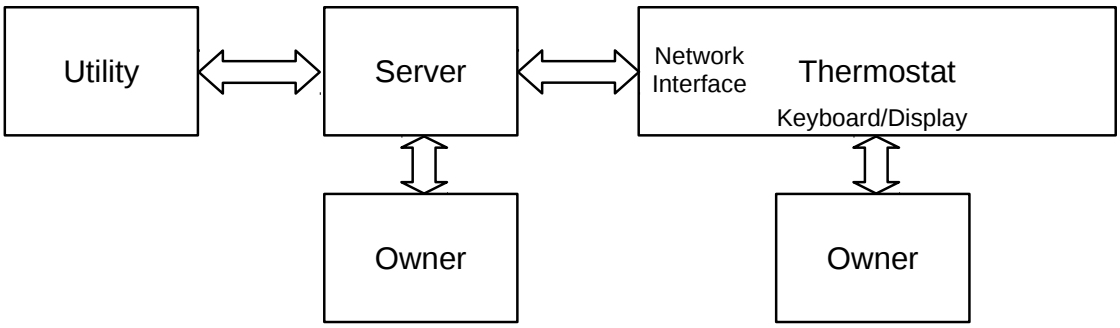
Security vulnerabilities often result from inconsistencies among CONOPS at various levels of abstraction. Military commanders might assume only authorized operators are able to launch an application, whereas the application itself might incorrectly trust that all orders it receives are from authorized operators and never authenticate the inputs it receives. Any *design for assurance* methodology must address authentication and authorization in order to *avoid vulnerabilities* due to unauthorized access or control. Rigorous assurance requires mathematical models and proofs. Our intent is illustrate a structured way to achieve security by design.

To illustrate the above concepts, throughout this chapter we apply them within the context of securing the integrity of a networked thermostat. We picked this example because (1) its function and purpose is easily understood, and (2) in a distributed control environment, its security and integrity concerns are representative of many other C2 applications.

1.2.2 A Networked Thermostat as a Motivating Example

Figure 1.2 shows a networked thermostat and its operating environment. The thermostat has a keyboard and a network interface. Commands received by the thermostat from its keyboard are assumed to originate from the thermostat's *Owner*. The *Owner* has the authority to execute any command.

Figure 1.2 A Networked Thermostat and Its Operating Environment



The thermostat also receives commands via a network interface to a remote *Server*. The *Server* relays commands from the *Owner* via the *Owner's* account on the *Server*. The *Server* relays commands from the *Utility* supplying energy to the *Owner*. The *Utility* has authority over the thermostat's operation if granted that authority by the *Owner*.

The reasons for granting authority to the *Utility* include reducing electrical loads on the grid during peak usage times. The benefits to the *Owner* are reduced electricity costs if cooling during the day can be deferred while the *Owner* is at work or away. The benefits to the *Utility* include deferred use of expensive generators as well as reduced strain on distribution systems.

Upon request, the thermostat reports its status back to the *Owner* and the *Utility* via the *Server* or using the physical display on the thermostat itself. The status of the thermostat is given by its *state*. Informally, the state of the thermostat is its operating *mode* and its *temperature* setting.

We consider three use-cases with respect to Figure 1.2.

1. The *Owner* issues commands via the thermostat's keyboard.
2. The *Owner* issues commands to the thermostat via the *Owner's* account on the *Server*.
3. The *Utility* issues commands to the thermostat via the *Server*.

At a high level, the thermostat commands are as follows.

1. **Setting** the temperature *value*. This command has security considerations as losing control over the temperature potentially is a threat to the safety of lives and property.
2. **Enabling** the *Utility* to exercise control over setting the temperature. This command has security considerations as *Owners* want to make sure they have the ultimate authority over their thermostat.
3. **Disabling** the *Utility* to exercise control over setting the temperature. This command has similar security considerations as the command used to enable the *Utility* to alter the thermostat's temperature setting.
4. Reporting the **Status** of the thermostat, which is displayed on the thermostat and sent to the *Server*. This command does not alter the thermostat's temperature setting or operating mode. As such, there are no security sensitivities with respect to reporting status. For our illustration, we assume there are no privacy concerns. If needed, privacy is handled in the usual ways including multi-level security, role-based access control, access-control lists, etc.

Thermostat Security CONOPS At this point in conceptualizing the networked thermostat, we need to consider the concepts we use to secure integrity of the thermostat's operations. We incorporate the following concepts into our design:

- Authenticating principals issuing commands using mechanisms such as (1) userids and passwords associated with *Owner* accounts on the *Server*, and (2) cryptographically signed messages from the *Server* to the thermostat and from the *Utility* to the *Server*,

- Authorizing principals issuing commands by making explicit the context in which authorization is done, i.e., public-key certificates, root trust assumptions on keys and jurisdiction, and policies stating what actions are taken in particular circumstances, and
- Executing or trapping commands based on a principal's authority and the security sensitivity of the command they are attempting to execute.

1.2.3 Assurance Requirements

The description of the networked thermostat example and the goals of CSBD lead us to the following requirements to realize the goals of CSBD.

- A C2 calculus used to reason about access-control decisions. The calculus we used is fully described in [8] and is an extension and modification of an access-control logic for distributed systems [4].
- Computer-assisted reasoning tools to (1) formally verify all proofs and assurance claims, and (2) enable rapid reproduction of all results by third parties and certifiers. We use the Cambridge University HOL-4 (Higher Order Logic) theorem prover [11]. It is freely available and has been in use since 1987.
- A model of idealized cryptographic operations and their properties implemented in HOL.
- Models of state machine transition systems incorporating authentication, authorization, next-state functions, and output functions as parameters in support of security and to avoid state explosion. Our networked thermostat illustration build upon the foundations of virtual machines, in particular [12].

1.3 Report Outline

The remainder of this report is organized as follows.

- Chapter 2 defines the syntax, semantics, and inference rules for an access-control logic used to reason about command and control (C2).
- Chapter 3 gives an overview of the Higher Order Logic (HOL) theorem prover we use as a computer-assisted reasoning (CAR) tool. The access-control logic is implemented as a conservative extension to HOL.
- Chapter 4 describes the HOL implementation of the access-control logic and the C2 calculus.
- Chapter 5 describes algebraic models of ideal cryptographic operations such as hashing, symmetric and asymmetric encryption, and cryptographic signing and verification. These algebraic models are implemented in HOL.
- Chapter 6 shows how security is built into state machines by labeled-transition descriptions incorporating security policies described in the access-control logic.
- Chapter 7 is a detailed example showing how security is designed into a networked thermostat.
- Chapter 8 contains our conclusions.

Chapter 2

An Access-Control Logic

This section describes an access-control logic that is our C2 calculus. Our description is brief for space considerations. A full account appears in [8]. We present the syntax, semantics, and inference rules in the following sections.

To follow the thermostat example, readers will need to comprehend the syntax and inference rules of the C2 calculus. Justifying the logical soundness of the C2 logic requires understanding the semantics of the logic. However, the semantics may be skipped if the primary purpose is to follow the thermostat example. Of course, the syntax, semantics, and inference rules are fully implemented and verified in HOL.

2.1 Syntax

The syntax of the logic has two major components.

1. The syntax of *principals*, where principals are informally thought of as the actors making statements, e.g., people, cryptographic keys, userids and passwords associated with accounts, etc.
2. The syntax of logical formulas.

The syntax of principal expressions *Princ* is defined as follows.

$$\textbf{Princ} \quad ::= \quad \textbf{PName} \;/\; \textbf{Princ} \;\&\; \textbf{Princ} \;/\; \textbf{Princ} \mid \textbf{Princ}$$

" & " is pronounced "with"; " | " is pronounced "quoting". The type of principal expressions is composed of principal names, e.g., Alice, cryptographic keys, and userid with passwords. Compound expressions are created with & and | .

Examples of principal expressions include

$$\mathit{Alice} \qquad K_{\mathit{Alice}} \qquad \mathit{Alice}\ \&\ \mathit{Bob} \qquad \mathit{Alice} \mid \mathit{Bob}$$

Informally, *Alice* is Alice, K_{Alice} is Alice's cryptographic key, *Alice* & *Bob* is Alice and Bob together, *Alice* | *Bob* is Alice quoting Bob (relaying his statements).

The syntax of logical formulas *Form* consists of propositional variables, expressions using the usual propositional operators corresponding to *modal* versions of negation, conjunction, disjunction, implication, and equivalence, coupled with operators $\Rightarrow$ (pronounced "speaks for"), *says*, *controls*, and *reps*.

In this presentation of the C2 calculus, we the same symbols for negation, conjunction, disjunction, implication, and equivalence in propositional logic. In the HOL implementation of the access-control logic, negation, conjunction, disjunction, implication, and equivalence in the access-control logic are represented using different symbols to clearly distinguish between access-control logic formulas and propositional logic formulas.

$$\begin{aligned}
\textbf{Form} \quad ::= \quad & \textbf{PropVar} \;/\; \neg\, \textbf{Form} \;/ \\
& (\textbf{Form} \vee \textbf{Form}) \;/\; (\textbf{Form} \wedge \textbf{Form}) \;/ \\
& (\textbf{Form} \supset \textbf{Form}) \;/\; (\textbf{Form} \equiv \textbf{Form}) \;/ \\
& (\textbf{Princ} \Rightarrow \textbf{Princ}) \;/\; (\textbf{Princ}\ \mathsf{says}\ \textbf{Form}) \;/ \\
& (\textbf{Princ}\ \mathsf{controls}\ \textbf{Form}) \;/\; \textbf{Princ}\ \mathsf{reps}\ \textbf{Princ}\ \mathsf{on}\ \textbf{Form}
\end{aligned}$$

Figure 2.1 is a table of typical C2 statements and their representation as formulas in the C2 calculus.

Figure 2.1 CONOPS Statements and Their Representation in the C2 Calculus

C2 Statement	Formula
If φ_1 is true then φ_2 is true (typical of policy statements)	$\varphi_1 \supset \varphi_2$
Key associated with Alice	$K_a \Rightarrow Alice$
Bob has jurisdiction (controls or is believed) over statement φ	Bob controls φ
Alice and Bob together say φ	$(Alice \ \& \ Bob)$ says φ
Alice quotes Bob as saying φ	$(Alice \mid Bob)$ says φ
Bob is Alice's delegate on statement φ	Bob reps $Alice$ on φ
Carol is authorized in Role on statement φ	$Carol$ reps $Role$ on φ
Carol acting in Role makes statement φ	$(Carol \mid Role)$ says φ

Figure 2.2 Kripke Semantics of Access-Control Logic Formulas

$$
\begin{aligned}
\mathcal{E}_{\mathcal{M}}[\![p]\!] &= I(p) \\
\mathcal{E}_{\mathcal{M}}[\![\neg\varphi]\!] &= W - \mathcal{E}_{\mathcal{M}}[\![\varphi]\!] \\
\mathcal{E}_{\mathcal{M}}[\![\varphi_1 \wedge \varphi_2]\!] &= \mathcal{E}_{\mathcal{M}}[\![\varphi_1]\!] \cap \mathcal{E}_{\mathcal{M}}[\![\varphi_2]\!] \\
\mathcal{E}_{\mathcal{M}}[\![\varphi_1 \vee \varphi_2]\!] &= \mathcal{E}_{\mathcal{M}}[\![\varphi_1]\!] \cup \mathcal{E}_{\mathcal{M}}[\![\varphi_2]\!] \\
\mathcal{E}_{\mathcal{M}}[\![\varphi_1 \supset \varphi_2]\!] &= (W - \mathcal{E}_{\mathcal{M}}[\![\varphi_1]\!]) \cup \mathcal{E}_{\mathcal{M}}[\![\varphi_2]\!] \\
\mathcal{E}_{\mathcal{M}}[\![\varphi_1 \equiv \varphi_2]\!] &= \mathcal{E}_{\mathcal{M}}[\![\varphi_1 \supset \varphi_2]\!] \cap \mathcal{E}_{\mathcal{M}}[\![\varphi_2 \supset \varphi_1]\!] \\
\mathcal{E}_{\mathcal{M}}[\![P \Rightarrow Q]\!] &= \begin{cases} W, & \text{if } \hat{J}(Q) \subseteq \hat{J}(P) \\ \emptyset, & \text{otherwise} \end{cases} \\
\mathcal{E}_{\mathcal{M}}[\![P \text{ says } \varphi]\!] &= \{w | \hat{J}(P)(w) \subseteq \mathcal{E}_{\mathcal{M}}[\![\varphi]\!]\} \\
\mathcal{E}_{\mathcal{M}}[\![P \text{ controls } \varphi]\!] &= \mathcal{E}_{\mathcal{M}}[\![(P \text{ says } \varphi) \supset \varphi]\!] \\
\mathcal{E}_{\mathcal{M}}[\![P \text{ reps } Q \text{ on } \varphi]\!] &= \mathcal{E}_{\mathcal{M}}[\![(P \mid Q \text{ says } \varphi) \supset Q \text{ says } \varphi]\!]
\end{aligned}
$$

2.2 Semantics

The semantics of the access-control logic uses *Kripke structures*. A **Kripke structure** $\mathcal{M}$ is a three-tuple $\langle W, I, J \rangle$, where:

- W is a nonempty set, whose elements are called *worlds*.
- I : **PropVar** $\rightarrow \mathcal{P}(W)$ is an *interpretation* function that maps each propositional variable p to a set of worlds.
- J : **PName** $\rightarrow \mathcal{P}(W \times W)$ is a function that maps each principal name A into a relation on worlds (i.e., a subset of $W \times W$).

The semantics of principal expressions *Princ* involves J and its extension $\hat{J}$. We define the extended function $\hat{J}$: **Princ** $\rightarrow \mathcal{P}(W \times W)$ inductively on the structure of principal expressions, where $A \in$ **PName**.

$$
\begin{aligned}
\hat{J}(A) &= J(A) \\
\hat{J}(P \ \& \ Q) &= \hat{J}(P) \cup \hat{J}(Q) \\
\hat{J}(P \mid Q) &= \hat{J}(P) \circ \hat{J}(Q).
\end{aligned}
$$

Note: $R_1 \circ R_2 = \{(x,z) \mid \exists y.(x,y) \in R_1 \text{ and } (y,z) \in R_2\}$.

Each Kripke structure $\mathcal{M} = \langle W, I, J \rangle$ gives rise to a **semantic function**

$$\mathcal{E}_{\mathcal{M}}[\![-]\!] : \textbf{Form} \rightarrow \mathcal{P}(W),$$

where $\mathcal{E}_{\mathcal{M}}[\![\varphi]\!]$ is the set of worlds in which φ is considered true.

$\mathcal{E}_{\mathcal{M}}[\![\varphi]\!]$ is defined inductively on the structure of φ, as shown in Figure 2.2. Note, in the definition of $\mathcal{E}_{\mathcal{M}}[\![P \text{ says } \varphi]\!]$, that $\hat{J}(P)(w)$ is simply the image of world w under the relation $\hat{J}(P)$.

Figure 2.3 Inference rules for the access-control logic

$$P \text{ controls } \varphi \stackrel{\text{def}}{=} (P \text{ says } \varphi) \supset \varphi \qquad P \text{ reps } Q \text{ on } \varphi \stackrel{\text{def}}{=} P \mid Q \text{ says } \varphi \supset Q \text{ says } \varphi$$

$$\textit{Modus Ponens}\ \frac{\varphi \qquad \varphi \supset \varphi'}{\varphi'} \qquad \textit{Says}\ \frac{\varphi}{P \text{ says } \varphi} \qquad \textit{Controls}\ \frac{P \text{ controls } \varphi \quad P \text{ says } \varphi}{\varphi}$$

$$\textit{Derived Speaks For}\ \frac{P \Rightarrow Q \quad P \text{ says } \varphi}{Q \text{ says } \varphi} \qquad \textit{Reps}\ \frac{Q \text{ controls } \varphi \quad P \text{ reps } Q \text{ on } \varphi \quad P \mid Q \text{ says } \varphi}{\varphi}$$

$$\&\ \textit{Says (1)}\ \frac{P \,\&\, Q \text{ says } \varphi}{P \text{ says } \varphi \wedge Q \text{ says } \varphi} \qquad \&\ \textit{Says (2)}\ \frac{P \text{ says } \varphi \wedge Q \text{ says } \varphi}{P \,\&\, Q \text{ says } \varphi}$$

$$\textit{Quoting (1)}\ \frac{P \mid Q \text{ says } \varphi}{P \text{ says } Q \text{ says } \varphi} \qquad \textit{Quoting (2)}\ \frac{P \text{ says } Q \text{ says } \varphi}{P \mid Q \text{ says } \varphi}$$

$$\textit{Idempotency of} \Rightarrow\ \frac{}{P \Rightarrow P} \qquad \textit{Monotonicity of} \Rightarrow\ \frac{P' \Rightarrow P \quad Q' \Rightarrow Q}{P' \mid Q' \Rightarrow P \mid Q}$$

2.3 Inference Rules

Our use of the access-control logic as a C2 calculus rarely, if ever, uses Kripke structures explicitly. Instead, we rely upon inference rules to derive expressions soundly.

An inference rule in the C2 calculus has the form

$$\frac{H_1 \quad \cdots \quad H_k}{C,}$$

where $H_1 \cdots H_k$ is a (possibly empty) set of *hypotheses* expressed as access-control logic formulas, and C is the *conclusion*, also expressed as an access-control logic formula. Whenever all of the hypotheses in an inference rule are present in a proof, then the rule states it is permissible to include the conclusion in the proof, too.

The meaning of *sound* depends on the the definition of *satisfies* in the access-control logic. A Kripke structure $\mathcal{M}$ **satisfies** a formula φ when $\mathcal{E}_{\mathcal{M}}[\![\varphi]\!] = W$, i.e., φ is true in all worlds W of $\mathcal{M}$. We denote $\mathcal{M}$ satisfies φ by $\mathcal{M} \models \varphi$.

A C2 calculus inference rule is **sound** if, for all Kripke structures $\mathcal{M}$, whenever $\mathcal{M}$ satisfies all the hypotheses $H_1 \cdots H_k$, then $\mathcal{M}$ also satisfies C, i.e., if for all $\mathcal{M}$: $\mathcal{M} \models H_i$ for $1 \leq i \leq k$, then it must be the case that $\mathcal{M} \models C$.

All the inference rules presented here and in [8] are proved to be logically sound. Figure 2.3 are the core inference rules of the access-control logic.

2.4 Describing Access-Control Concepts in the C2 Calculus

To illustrate how the C2-calculus is used to reason about authentication and authorization, we consider the following use case.

Example 2.4.1

Bob guards access to sensitive files. He receives requests electronically and says yes or no to each request. Specifically, the requests he receives are are digitally signed by a cryptographic key. Keys are associated with people, e.g., Alice. If the person, say Alice, who owns the key has permission to access the file, then Bob says yes.

Suppose Bob receives an access request signed by Alice's key K_A, and that Alice is permitted to access the files. We represent the request, the link between Alice and her key K_A, and her permission to access the files by the following statements in the access-control logic.

1. Digitally signed request received by Bob: K_A says $\langle access\ files \rangle$.
2. K_A is Alice's key: $K_A \Rightarrow Alice$.

3. Alice has permission to access the files: $Alice$ controls $\langle access\ files \rangle$

Using the inference rules of the C2 calculus, Bob justifies his decision to grant Alice's request by the following proof, where lines 1–3 are the assumptions, and everything that follows is derived using the inference rules of the C2 calculus.

1.	K_A says $\langle access\ files \rangle$	Digitally signed request
2.	$K_A \Rightarrow Alice$	Key associated with Alice
3.	$Alice$ controls $\langle access\ files \rangle$	Alice's capability to access files
4.	$Alice$ says $\langle access\ files \rangle$	2, 1 Derived Speaks For
5.	$\langle access\ files \rangle$	3, 4 Controls

Line 4 amounts to authenticating that Alice is the originator of the access request within the context established by lines 1 through 3. Line 3 establishes Alice's authority to access the files. Line 5 is Bob's deduction that granting Alice access is justified.

As a result of the proof, Bob has a derived inference rule, which he knows is sound because he derived it using the inference rules in Figure 2.3. The derived inference rule is

$$\frac{K_A \text{ says } \langle access\ files \rangle \qquad K_A \Rightarrow Alice \quad Alice \text{ controls } \langle access\ files \rangle}{\langle access\ files \rangle}$$

The inference rule amounts to a checklist. If he (1) gets a message cryptographically signed with K_A, (2) K_A is Alice's key, and (3) Alice has permission to access the files, then granting access to Alice is justified.

Looking back at Figure 1.1, the inference rule is a logically sound description of what Bob does in the top-level CONOPS. The inference rule makes explicit the policies and trust assumptions and how they combine to justify Bob's actions. ◊

Delegation is widely used. Our definition of delegation is given by the definition of *reps* and the *Reps* inference rule.

$$P \text{ reps } Q \text{ on } \varphi \stackrel{\text{def}}{=} P \mid Q \text{ says } \varphi \supset Q \text{ says } \varphi$$

$$Reps \quad \frac{Q \text{ controls } \varphi \quad P \text{ reps } Q \text{ on } \varphi \quad P \mid Q \text{ says } \varphi}{\varphi}$$

The consequence of the definition of *reps* in the first formula shows is this: if you believe $Alice$ reps Bob on φ is true, then if Alice says Bob says φ you will conclude that Bob says φ. In other words, Alice is trusted when she says Bob says φ.

In a command and control application, if you believe (1) Bob is authorized on command φ, (2) Alice is Bob's delegate or representative on a command φ, and (3) Alice says Bob says command φ, then you are justified to conclude the command φ is legitimate. This is the *Reps* inference rule.

Reps is particularly useful for delegating limited authority to delegates. Unlike $\Rightarrow$, where all statements of one principal are attributable to another, *Reps* specifies which statements made by a delegate are attributable to another.

Reps is used when people are acting in defined roles, for example the roles of *Commander* and *Operator*. The following example show the use of *reps* in the context of roles.

Example 2.4.2

Suppose we have two roles, two people, and two commands. The roles are *Commander* and *Operator*; the people are Alice and Bob; the two commands are *go* and *launch*. A *Commander* has the authority to issue a *go* command. An *Operator* has the authority to issue a *launch* command whenever a *go* command is received from a *Commander*. *Commanders* are not authorized to *launch*. *Operators* are not authorized to *launch* unless they receive a *go* command.

In this scenario, *Alice* is the *Commander* and *Bob* is an *Operator*. Notice that this scenario is captured by Figure 1.1.

We represent the notion that *Alice* and *Bob* are acting in their assigned roles of *Commander* and *Operator* using quotation and delegation. With Figure 1.1 in mind, we do the following analysis from Bob's perspective.

1. Message Bob receives signed by Alice's key:

$$K_A \mid Commander \text{ says } \langle go \rangle$$

2. Bob's belief that K_A is Alice's key:

$$K_A \Rightarrow Alice$$

3. Bob's recognition that Alice is acting as *Commander* when issuing a *go* command:

$$Alice \text{ reps } Commander \text{ on } \langle go \rangle.$$

4. Bob's belief that *Commanders* have authority to issue *go* commands:

$$Commander \text{ controls } \langle go \rangle.$$

5. The policy guiding Bob's actions, when he authenticates and authorizes a *go* command, then he is to issue a *launch* command:

$$\langle go \rangle \supset \langle launch \rangle$$

The input in line 1 with the other 4 assumptions as security context for Bob's decision is sufficient for Bob to issue the command $K_B \mid Operator \text{ says } \langle launch \rangle$. The proof is as follows using the inference rules in Figure 2.3.

1.	$K_A \mid Commander \text{ says } \langle go \rangle$	Input signed by K_A
2.	$K_A \Rightarrow Alice$	Trust assumption—K_A is Alice's key
3.	$Alice \text{ reps } Commander \text{ on } \langle go \rangle$	Trust assumption—Alice is acting as a Commander when issuing a *go* command
4.	$Commander \text{ controls } \langle go \rangle$	Trust assumption—Commanders have authority to issue a *go* command
5.	$\langle go \rangle \supset \langle launch \rangle$	Policy assumption—if *go* is true then so is *launch*
6.	$Commander \Rightarrow Commander$	Idempotency of $\Rightarrow$
7.	$K_A \mid Commander \Rightarrow Alice \mid Commander$	2, 6 Monotonicity of $\Rightarrow$
8.	$Alice \mid Commander \text{ says } \langle go \rangle$	7, 1 Derived Speaks For
9.	$\langle go \rangle$	4, 3, 8 Reps
10.	$\langle launch \rangle$	9, 5 Modus Ponens
11.	$K_B \mid Operator \text{ says } \langle launch \rangle$	10 Says

The above proof justifies a derived inference rule showing the soundness of Bob's actions:

$$\frac{\begin{array}{c} K_A \mid Commander \text{ says } \langle go \rangle \\ K_A \Rightarrow Alice \quad Alice \text{ reps } Commander \text{ on } \langle go \rangle \\ Commander \text{ controls } \langle go \rangle \quad \langle go \rangle \supset \langle launch \rangle \end{array}}{K_B \mid Operator \text{ says } \langle launch \rangle}$$

The derived inference rule is a logical checklist. If (1) *Bob* receives a cryptographically signed message using key K_A issuing a *go* order while quoting a *Commander* role, (2) K_A is Alice's key, (3) *Alice* is authorized to issue a *go* command as a *Commander*, (4) *Commanders* have the authority to issue a *go* command, and (5) the policy is when *go* is true the *launch* is true, then issuing $K_B \mid Operator \text{ says } \langle launch \rangle$ is justified, where K_B is Bob's key. $\Diamond$

We now turn our attention to automated support for reasoning using the HOL theorem prover for the C2 calculus in the next section, and cryptographic operations and for state transition systems in subsequent sections.

Chapter 3

An Introduction to HOL

Automated tools are essential for any realistic design and verification methodology. In this section, we introduce our use of the HOL theorem prover [11]. A detailed description is infeasible given space limitations, and is beyond the scope of this chapter. Instead, we present an introduction to how proofs are done in HOL with enough detail to enable a reading level of comprehension. The HOL system is equipped with several tutorials, user guides, and encyclopedic manuals. HOL and its documentation are available freely from online sources[1].

The advantages of using computer-assisted reasoning (CAR) tools in general and HOL specifically include:

1. formal verification of assurance claims,
2. automated support to manage large and complicated formulas and proofs,
3. access to vast and comprehensive libraries of verified theories containing definitions and theorems spanning mathematical logic, programming languages, instruction sets, and microprocessors, allowing designers to easily build upon a logically sound foundation of previous work,
4. LaTeX macros of definitions, theorems, and formulas automatically generated by HOL, thus reducing or eliminating the burden of manually typesetting formulas and introducing typographical errors, while enabling easy updates to documentation when theories are modified, and
5. rapid and easy reproduction by third parties of all verification results.

All of the above factors combine to produce *precision, accuracy, and confidence* in assurance results. Results verified in HOL enable (1) system designers and verifiers to have confidence in their own work, and (2) others with more technical sophistication and experience, to reproduce and have confidence in results produced by those with comparatively less experience and sophistication.

In the following three examples, we define two parameterized theories of state machines and show they are equivalent. In the next section, we show the syntax, semantics, and HOL theorems that define the access-control logic in HOL.

Example 3.0.1

Suppose we wish to define state machines parametrically in terms of their state, input, and next-state transition functions, as shown in Figure 3.1. States and inputs are envisioned to be any type and each may have an infinite number of elements. The notation in Figure 3.1 is used in HOL. Terms and their types are represented in HOL by $hol\_term : hol\_type$, i.e., HOL terms followed by their types separated by a colon. For example $1 : num$, states that 1 is of type *num* in HOL.

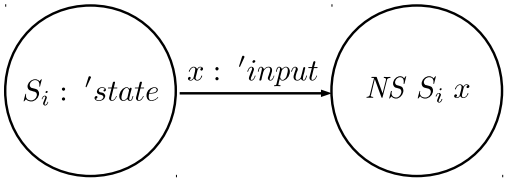
HOL supports polymorphism by using *type variables*. Type variables in HOL all have a leading prime symbol, $'$. Figure 3.1 shows state S_i with type variable $'state$, symbolized by $S_i : 'state$. The expression $x : 'input$ says x is polymorphic with type variable $'input$, which also can be any type. As $'state$ and $'input$ are different type variables, the types of S_i and x need not (and typically are not) the same.

The state-transition behavior of a deterministic state-machine is defined by its next-state function. In Figure 3.1 this is the function *NS*. What the figure shows is that if the machine is in state S_i, then the next state of the machine is $NS\ S_i\ x$, where the type signature of *NS* is $'state \rightarrow' input \rightarrow' state$.

The arrow labeled with $x : 'input$ from state S_i to state $NS\ S_i\ x$ is modeled as an *inductively defined relation* in HOL. Inductive relations are used to define familiar sets of objects, for example the set of *even* numbers. The set of even numbers is specified by the following rules:

[1] Readers who are interested in using HOL are able to download its sources and executable images from sites easily found by common search engines

Figure 3.1 Parameterized State-Transition Relation



1. 0 is **even**.
2. If n is **even** then $n+2$ is **even**.
3. The set **even** is the *smallest set* satisfying rules (1) and (2).

HOL has an extensive library of theories and functions, including functions for inductive definitions. The code snippet below illustrates the use `Hol_reln` to define inductively the predicate *even* on the natural numbers. The function `Hol_reln` when applied to its arguments corresponding to rules (1) and (2) above, returns three theorems, which are assigned to names *even_rules*, *even_induction*, and *even_cases*. In HOL, `val` is used for assigning values to names. HOL supports pattern matching, so we can assign names to the 3-tuple of theorems returned by `Hol_reln`. (Note: HOL uses ASCII symbols; `!` is the universal quantifier $\forall$; `==>` is logical implication $\Rightarrow$; and `/\` is logical conjunction $\wedge$.

```
val (even_rules, even_induction, even_cases) =
Hol_reln
 `even 0 /\
 (!n. even n ==> even (n + 2))`;
```

The HOL code above produces three theorems, which are pretty-printed below, using HOL-generated LaTeX macros. HOL uses *sequents* to represent theorems. Sequents have the form $\Gamma \vdash t$, where t is a term in predicate logic and Γ is a set of predicate logic terms. What $\Gamma \vdash t$ states is when all the terms in Γ are true, then t must be true, too. If Γ is empty, then we write $\vdash t$. In each of the following three theorems, Γ is empty.

```
[even_rules]
⊢ even 0 ∧ ∀n. even n ⇒ even (n + 2)

[even_induction]
⊢ ∀even'.
      even' 0 ∧ (∀n. even' n ⇒ even' (n + 2)) ⇒
      ∀a0. even a0 ⇒ even' a0

[even_cases]
⊢ ∀a0. even a0 ⇔ (a0 = 0) ∨ ∃n. (a0 = n + 2) ∧ even n
```

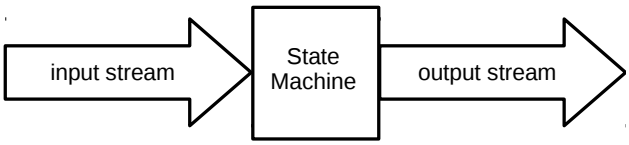
The first theorem *even_rules* is a commonly used description of even numbers: 0 is even, and if n is even then so is $n+2$. The second theorem *even_induction* is an induction principle using the fact that the inductive definition of *even* is the *smallest set of numbers satisfying the even_rules*. In other words, if a relation *even*$'$ satisfies the same rules as *even*, then when *even* is true *even*$'$ must be true, too. Finally, the third theorem *even_cases* states that if a_0 is even, then a_0 is either 0 or there is an even number n such that $a_0 = n+2$.

Returning to formalizing what is expressed graphically in Figure 3.1 by $\xrightarrow{x}$, we define a labeled transition relation *Trans x*, in words as follows.

1. For all next-state functions *NS*, inputs x, and states s, the predicate *Trans x* is true for states s and *NS s x*.
2. The set defining *Trans x* is the smallest set satisfying rule (1).

The following code snippet defines the transition relation *Trans* labeled with input x.

Figure 3.2 State Machine Behavior with Input and Output Streams




```
val (Trans_rules, Trans_ind, Trans_cases) =
Hol_reln
‘!NS (s:’state) (x:’input).
  Trans x s ((NS:’state -> ’input -> ’state) s x)‘
```

`Hol_reln` returns three theorems, *Trans_rules*, *Trans_ind*, and *Trans_cases* shown below.

```
[Trans_rules]
⊢ ∀NS s x. Trans x s (NS s x)

[Trans_ind]
⊢ ∀Trans'.
      (∀NS s x. Trans' x s (NS s x)) ⇒
      ∀a0 a1 a2. Trans a0 a1 a2 ⇒ Trans' a0 a1 a2

[Trans_cases]
⊢ ∀a0 a1 a2. Trans a0 a1 a2 ⟺ ∃NS. a2 = NS a1 a0
```

The first theorem *Trans_rules* is a formalization of rule (1). The second theorem *Trans_ind* is a consequence of *Trans x* being the small set satisfying rule (1). *Trans_cases* state that for all inputs a_0 and states a_1 and a_2, there is always some next-state function *NS* such that a_2 is the next state of a_1 for a given input a_0.

While the above example is simple, it shows some of the advantages of higher-order logic in general, and computer-assisted reasoning tools, such as HOL, in particular. The higher-order nature of the logic allows us to parameterize over functions, e.g., the next-state function *NS*. HOL's extensive library of theories and functions supports the creation of logically sound extensions by engineers. ◊

Example 3.0.2

In this example we define another way of looking at the behavior of state machines. Figure 3.2 shows a state machine with both an input stream and an output stream. Both streams are modeled by lists of inputs and outputs. The state machine is described parametrically by a next-state function *NS*.

We can define a transition relation *TR x* similar to the relation *Trans x* in Example 3.0.1, except this relation is over *state-machine configurations* that incorporate input streams, state, and output streams. We define a *configuration* algebraic type in HOL using the code snippet below.

```
val _ =
Hol_datatype
‘configuration =
  CFG of ’input list => ’state => ’output list‘
```

The HOL function `Hol_datatype` introduces new type definition into HOL. In this case, the datatype *configuration* is defined as having a type constructor `CFG` and takes as inputs three arguments whose types are $'input\ list$, $'state$, and $'output\ list$. These arguments are polymorphic, as indicated by their respective type variables, and correspond to input streams, states, and output streams. The pretty-printed result of executing the above code snippet is the introduction of *configuration* as an algebraic type.

```
configuration = CFG ('input list) 'state ('output list)
```

Conveniently, HOL provides extensive support for reasoning about algebraic types. In particular, we use the HOL function `one_one_of` to prove a theorem stating that two configurations are equal if and only if their components are equal. The code snippet is shown below along with the pretty-printed theorem `configuration_one_one`.

```
val
 configuration_one_one =
 one_one_of ``:('input,'state,'output)configuration``
```

```
[configuration_one_one]
⊢ ∀a0 a1 a2 a0' a1' a2'.
      (CFG a0 a1 a2 = CFG a0' a1' a2') ⟺
      (a0 = a0') ∧ (a1 = a1') ∧ (a2 = a2')
```

With the algebraic type *configuration* defined in HOL, we define the relation *TR x* on a starting configuration whose input stream is *x::ins*, state *s*, and output stream *outs*, with next-state transition function *NS* and output function *Out*.

1. For all next-state functions *NS*, output functions *Out*, inputs *x*, input streams *ins*, states *s*, and output streams *outs*, the predicate *TR x* is true for configurations *(CFG (x::ins) s outs)* and *(CFG ins (NS s x) (Out s x::outs))*.

2. The set defining *TR x* is the smallest set satisfying rule (1).

The code snippet below defines the transition relation *TR x* on configurations with input *x*.

```
val (TR_rules, TR_ind, TR_cases) =
Hol_reln
``!NS Out (s:'state) (x:'input) (ins:'input list)
  (outs:'output list).
  TR x
   (CFG (x::ins) s outs)
   (CFG ins (NS s x) ((Out s x)::outs))``
```

`Hol_reln` returns three theorems, *TR_rules*, *TR_ind*, and *TR_cases* shown below.

```
[TR_rules]
⊢ ∀NS Out s x ins outs.
      TR x (CFG (x::ins) s outs)
        (CFG ins (NS s x) (Out s x::outs))

[TR_ind]
⊢ ∀TR'.
      (∀NS Out s x ins outs.
         TR' x (CFG (x::ins) s outs)
           (CFG ins (NS s x) (Out s x::outs))) ⇒
      ∀a0 a1 a2. TR a0 a1 a2 ⇒ TR' a0 a1 a2

[TR_cases]
⊢ ∀a0 a1 a2.
      TR a0 a1 a2 ⟺
      ∃NS Out s ins outs.
        (a1 = CFG (a0::ins) s outs) ∧
        (a2 = CFG ins (NS s a0) (Out s a0::outs))
```

As with similar definitions, *TR_rules* is the formalization of rule (1), *TR_ind* is a result of *TR x* being the smallest set satisfying rule (1), and *TR_cases* relates the components of the second configuration to the components of the first configuration in conjunction with the next state and output functions *NS* and *Out*, respectively. ◊

Example 3.0.3

With two definitions of transition relations on state machines, we can prove they are logically equivalent. In this example, we give a brief illustration of goal-oriented proof in HOL. The theorem we prove as an illustration states that if *Trans x s (NS s x)* is true, then so is *TR x (CFG (x::ins) s outs) (CFG ins (NS s x) (Out s x::outs))*. The theorem below, *Trans_TR_lemma* states this fact.

```
[Trans_TR_lemma]
 ⊢ Trans x s (NS s x) ⇒
    TR x (CFG (x::ins) s outs) (CFG ins (NS s x) (Out s x::outs))
```

In HOL, goal-oriented proofs work by stating the desired goal with the same components as a sequent corresponding to ultimate theorem: we provide a pair consisting of a list of assumptions and the conclusion. This is done by the HOL function `set_goal`. Below, `set_goal` is applied to `([],``(Trans (x:'input) (s:'state) (NS s x))`, i.e., the goal of proving *Trans x* implies *TR x*, with no assumptions. `set_goal` returns the proof state of proving *Trans x* implies *TR x* with no assumptions.

```
- set_goal([],``(Trans (x:'input) (s:'state) (NS s x)) ==>
(TR x (CFG (x::ins) s (outs:'output list))(CFG ins (NS s x) ((Out s x)::outs)))``);
> val it =
    Proof manager status: 1 proof.
    1. Incomplete goalstack:
         Initial goal:

         Trans x s (NS s x) ==>
         TR x (CFG (x::ins) s outs) (CFG ins (NS s x) (Out s x::outs))
```

Our next proof step is to simplify the assumptions as much as possible by moving all antecedents of implications into the assumption list. This is done by executing `STRIP_TAC`.

```
- e(STRIP_TAC);
OK..
1 subgoal:
> val it =

    TR x (CFG (x::ins) s outs) (CFG ins (NS s x) (Out s x::outs))
    ------------------------------------
      Trans x s (NS s x)
     : proof
```

We recognize that the goal corresponds to the theorem *TR_rules*. We supply *TR_rules* to a high-level decision procedure in HOL name `PROVE_TAC`. The results and completed proof are below.

```
- e(PROVE_TAC[TR_rules]);
OK..
Meson search level: ..

Goal proved.
 [.] |- TR x (CFG (x::ins) s outs) (CFG ins (NS s x) (Out s x::outs))
> val it =
    Initial goal proved.
    |- Trans x s (NS s x) ==>
       TR x (CFG (x::ins) s outs) (CFG ins (NS s x) (Out s x::outs))
     : proof
```

In a similar fashion, we prove the converse of `Trans_TR_lemma`. The theorem is shown below as *TR_Trans_lemma*.

```
[TR_Trans_lemma]
 ⊢ TR x (CFG (x::ins) s outs)
       (CFG ins (NS s x) (Out s x::outs)) ⇒
    Trans x s (NS s x)
```

With the two lemmas *Trans_TR_lemma* and *TR_Trans_lemma*, it is straightforward to prove that *Trans* and *TR* are logically equivalent. The following code snippet illustrates how the HOL function `TAC_PROOF` is used to prove the logical equivalence of *Trans* and *TR*.

```
val Trans_Equiv_TR =
TAC_PROOF
(([],
``(TR (x:'input)
   (CFG (x::ins) (s:'state)(outs:'output list))
   (CFG ins (NS s x)((Out s x)::outs))) =
  (Trans (x:'input) (s:'state) (NS s x))``),
PROVE_TAC[TR_Trans_lemma,Trans_TR_lemma])
```

The results of the proof are shown below.

```
- val Trans_Equiv_TR =
TAC_PROOF(
([],
``(TR (x:'input)
   (CFG (x::ins) (s:'state)(outs:'output list))
   (CFG ins (NS s x)((Out s x)::outs))) =
  (Trans (x:'input) (s:'state) (NS s x))``),
PROVE_TAC[TR_Trans_lemma,Trans_TR_lemma]);
Meson search level: ......
> val Trans_Equiv_TR =
    |- TR x (CFG (x::ins) s outs) (CFG ins (NS s x) (Out s x::outs)) <=>
       Trans x s (NS s x)
     : thm
```

♢

The three examples in this section briefly illustrate how definitional extension and proofs are done in HOL. In the remaining sections, we focus on the definitions and theorems, while omitting the details of how the proofs are done in HOL.

Note: in everything that follows, all formulas starting with $\vdash$ are theorems in HOL, typeset in LaTeX by HOL, and formally verified in HOL.

Chapter 4

The Access-Control Logic in HOL

The access-control logic described in Chapter 2 is implemented in HOL by (1) defining its syntax as an algebraic type *Form*, (2) inductively defining the semantic function $\mathcal{E}_{\mathcal{M}}[\![-]\!]$ in HOL over the type *Form* of access-control logic formulas, and (3) proving theorems in HOL corresponding to inference rules of the C2 calculus.

The benefits of implementing the access-control logic in HOL include:

1. complete disclosure of all access-control logic and C2 calculus syntax and semantics,
2. formal machine-checked proofs of all properties of the access-control logic,
3. quantification over access-control logic formulas,
4. ability to combine the access-control logic with other logical descriptions, and
5. rapid and easy reproduction of all results by third parties.

Sections 4.1, 4.2, and 4.3 describe the syntax, semantics, and theorems corresponding to the inference rules of the access-control logic and C2 calculus, respectively.

4.1 Syntax of the Access-Control Logic in HOL

The access-control logic is implemented as a conservative extension to the HOL system. What this means is that the HOL logic is extended by defining the *Form* algebraic type corresponding to access-control logic formulas, the algebraic type *Princ* corresponding to principal expressions, and the algebraic type *Kripke* corresponding to Kripke structures. The semantics of *Form* and *Princ* are defined using *Kripke* and existing HOL operators. The properties of the access-control logic are proved as theorems in HOL.

Figure 4.1 shows the HOL type *Form* corresponding to access-control logic formulas in HOL. Notice that the HOL implementation uses *notf, andf, orf, impf* and *eqf* to represent negation, conjunction, disjunction, implication , and equivalence in the access-control logic. Their semantics is defined in terms of sets of worlds from the universe of worlds that is part of a Kripke structure $\mathcal{M}$. This is different than the semantics of the corresponding operators in propositional logic. The propositional logic operators are defined in terms of truth values instead of sets of worlds.

The type definition in Figure 4.1 is polymorphic, i.e., allows for type substitution into type variables. Recall that type variables in HOL start with the back-quote symbol `'`. For example, atomic propositions in the access-control logic in HOL start with the type constructor `prop` and are applied to any type, as represented by `'aavar`. For example, `prop command` takes elements of the type `command` and maps them to propositions in the access-control logic in HOL.

Figure 4.2 shows the syntax of principal expressions, integrity and security labels, and Kripke structures in HOL. The HOL implementation parameterizes security labels, integrity labels, and their partial orders. As our thermostat example does not rely upon security or integrity labels, we will not discuss their use further. Examples using security and integrity labels are in [8].

The type constructor `Name` is polymorphic as seen in the type definition of *Princ*, where it is applied to the type variable `'apn`. The infix type constructor `meet` corresponds to & . The infix type constructor `quoting` corresponds to | .

Figure 4.3 is a table showing how formulas in the C2 calculus are written in HOL implementation of the access-control logic. The proposition $\langle jump \rangle$ is written as `prop jump` in HOL. Negation of a C2 formula, such as $\neg\langle jump \rangle$ is written as `notf` (`prop jump`) in HOL. *Alice* says $\langle jump \rangle$ is written as `Name Alice says prop jump`, etc.

Figure 4.1 Access-Control Logic Syntax in HOL

```
Form =
    TT
  | FF
  | prop 'aavar
  | notf (('aavar, 'apn, 'il, 'sl) Form)
  | (andf) (('aavar, 'apn, 'il, 'sl) Form)
           (('aavar, 'apn, 'il, 'sl) Form)
  | (orf) (('aavar, 'apn, 'il, 'sl) Form)
          (('aavar, 'apn, 'il, 'sl) Form)
  | (impf) (('aavar, 'apn, 'il, 'sl) Form)
           (('aavar, 'apn, 'il, 'sl) Form)
  | (eqf) (('aavar, 'apn, 'il, 'sl) Form)
          (('aavar, 'apn, 'il, 'sl) Form)
  | (says) ('apn Princ) (('aavar, 'apn, 'il, 'sl) Form)
  | (speaks_for) ('apn Princ) ('apn Princ)
  | (controls) ('apn Princ) (('aavar, 'apn, 'il, 'sl) Form)
  | reps ('apn Princ) ('apn Princ)
         (('aavar, 'apn, 'il, 'sl) Form)
  | (domi) (('apn, 'il) IntLevel) (('apn, 'il) IntLevel)
  | (eqi) (('apn, 'il) IntLevel) (('apn, 'il) IntLevel)
  | (doms) (('apn, 'sl) SecLevel) (('apn, 'sl) SecLevel)
  | (eqs) (('apn, 'sl) SecLevel) (('apn, 'sl) SecLevel)
  | (eqn) num num
  | (lte) num num
  | (lt) num num
```

4.2 Semantics of the Access-Control Logic in HOL

With the introduction of logical expressions, principal expressions, and Kripke structures as datatypes into HOL, we can define the HOL function `Efn` corresponding to the function $\mathcal{E}_{\mathcal{M}}[\![-]\!]$ in Figure 2.2, which defines the Kripke semantics of the access-control logic. The definition of `Efn` is in Section B. The definitions of $\mathcal{E}_{\mathcal{M}}[\![-]\!]$ and `Efn` closely correspond to one another syntactically.

Of course, the question is how do we know that the implementation in HOL corresponds to the logic described in Figure 2.2 and as described in [8]? The answer is if we can prove theorems in HOL about the HOL implementation that correspond to the inference rules in [8], then we are satisfied.

4.3 C2 Inference Rules in HOL

Recall in Section 2.3 that $\mathcal{M} \models \varphi$ denoted $\mathcal{E}_{\mathcal{M}}[\![\varphi]\!] = W$, i.e., φ is true for all worlds in $\mathcal{M}$. Inference rules in the C2 calculus are sound because whenever $\mathcal{M}$ satisfies all the hypotheses $H_1 \cdots H_k$, then $\mathcal{M}$ satisfies conclusion C as well.

In our HOL implementation, we say Kripke structure M with partial orders O_i and O_s on integrity and security labels, respectively, satisfies an access-control logic formula f whenever the HOL semantic function `Efn`, whose definition appears in Section B, applied to M, O_i, O_s, and f equals the universe of worlds in M. The definition of `sat` in HOL is as follows.

```
[sat_def]
⊢ ∀M Oi Os f. (M,Oi,Os) sat f ⟺ (Efn Oi Os M f = 𝒰(:'world))
```

An inference rule in the C2 calculus of the form

$$\frac{H_1 \cdots H_k}{C}$$

Figure 4.2 Syntax of Principal Expressions, Integrity and Security Labels, and Kripke Structures in HOL

```
Princ =
    Name 'apn
  | (meet) ('apn Princ) ('apn Princ)
  | (quoting) ('apn Princ) ('apn Princ) ;

IntLevel = iLab 'il | il 'apn ;

SecLevel = sLab 'sl | sl 'apn



Kripke =
    KS ('aavar -> 'aaworld -> bool)
       ('apn -> 'aaworld -> 'aaworld -> bool) ('apn -> 'il)
       ('apn -> 'sl)
```

Figure 4.3 C2 Formulas and Their Representation in HOL

C2 Formula	**HOL Syntax**
$\langle jump \rangle$	`prop jump`
$\neg\langle jump \rangle$	`notf (prop jump)`
$\langle run \rangle \wedge \langle jump \rangle$	`prop run andf prop jump`
$\langle run \rangle \vee \langle stop \rangle$	`prop run orf prop stop`
$\langle run \rangle \supset \langle jump \rangle$	`prop run impf prop jump`
$\langle walk \rangle \equiv \langle stop \rangle$	`prop walk eqf prop stop`
Alice says $\langle jump \rangle$	`Name Alice says prop jump`
Alice & *Bob* says $\langle stop \rangle$	`Name Alice meet Name Bob says prop stop`
Bob | *Carol* says $\langle run \rangle$	`Name Bob quoting Name Carol says prop run`
Bob controls $\langle walk \rangle$	`Name Bob controls prop walk`
Bob reps *Alice* on $\langle jump \rangle$	`reps (Name Bob) (Name Alice) (prop jump)`
Carol $\Rightarrow$ *Bob*	`Name Carol` speaks_for `Name Bob`

has a corresponding theorem in HOL

$$\vdash \forall M\ O_i\ O_s.(M, O_i, O_s)\ sat\ H_1 \Rightarrow \cdots \Rightarrow (M, O_i, O_s)\ sat\ H_k \Rightarrow (M, O_i, O_s)\ sat\ C,$$

where $\Rightarrow$ corresponds to logical implication in HOL. Figures 4.4 and 4.5 show the HOL theorems corresponding to the C2 inference rules in Figure 2.3.

Figure 4.4 HOL Theorems Corresponding to C2 Calculus Inference Rules (1 of 2)

[Controls_Eq]

$\vdash \forall M\ Oi\ Os\ P\ f.$
$\quad (M, Oi, Os)\ \texttt{sat}\ P\ \texttt{controls}\ f \iff (M, Oi, Os)\ \texttt{sat}\ P\ \texttt{says}\ f\ \texttt{impf}\ f$

[Reps_Eq]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
$\quad (M, Oi, Os)\ \texttt{sat}\ \texttt{reps}\ P\ Q\ f \iff$
$\quad (M, Oi, Os)\ \texttt{sat}\ P\ \texttt{quoting}\ Q\ \texttt{says}\ f\ \texttt{impf}\ Q\ \texttt{says}\ f$

[Modus Ponens]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2.$
$\quad (M, Oi, Os)\ \texttt{sat}\ f_1 \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ f_1\ \texttt{impf}\ f_2 \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ f_2$

[Says]

$\vdash \forall M\ Oi\ Os\ P\ f.\ (M, Oi, Os)\ \texttt{sat}\ f \Rightarrow (M, Oi, Os)\ \texttt{sat}\ P\ \texttt{says}\ f$

[Controls]

$\vdash \forall M\ Oi\ Os\ P\ f.$
$\quad (M, Oi, Os)\ \texttt{sat}\ P\ \texttt{says}\ f \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ P\ \texttt{controls}\ f \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ f$

[Derived_Speaks_For]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
$\quad (M, Oi, Os)\ \texttt{sat}\ P\ \texttt{speaks\_for}\ Q \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ P\ \texttt{says}\ f \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ Q\ \texttt{says}\ f$

Figure 4.5 HOL Theorems Corresponding to C2 Calculus Inference Rules (2 of 2)

[Reps]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
$\quad (M, Oi, Os)\ \texttt{sat reps}\ P\ Q\ f \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ P\ \texttt{quoting}\ Q\ \texttt{says}\ f \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ Q\ \texttt{controls}\ f \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ f$

[And_Says_Eq]

$\vdash (M, Oi, Os)\ \texttt{sat}\ P\ \texttt{meet}\ Q\ \texttt{says}\ f \iff$
$\quad (M, Oi, Os)\ \texttt{sat}\ P\ \texttt{says}\ f\ \texttt{andf}\ Q\ \texttt{says}\ f$

[Quoting_Eq]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
$\quad (M, Oi, Os)\ \texttt{sat}\ P\ \texttt{quoting}\ Q\ \texttt{says}\ f \iff$
$\quad (M, Oi, Os)\ \texttt{sat}\ P\ \texttt{says}\ Q\ \texttt{says}\ f$

[Idemp_Speaks_For]

$\vdash \forall M\ Oi\ Os\ P.\ (M, Oi, Os)\ \texttt{sat}\ P\ \texttt{speaks\_for}\ P$

[Mono_Speaks_For]

$\vdash \forall M\ Oi\ Os\ P\ P'\ Q\ Q'.$
$\quad (M, Oi, Os)\ \texttt{sat}\ P\ \texttt{speaks\_for}\ P' \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ Q\ \texttt{speaks\_for}\ Q' \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ P\ \texttt{quoting}\ Q\ \texttt{speaks\_for}\ P'\ \texttt{quoting}\ Q'$

Chapter 5

Cryptographic Components and Their Models in Higher Order Logic

Cryptographic operations are an integral part of protecting integrity and confidentiality. In this section, we provide algebraic models in higher-order logic and HOL of idealized cryptographic operations. Missing is any notion of cryptographic strength and a particular algorithm's ability to withstand cryptanalysis.

Our descriptions of ideal cryptographic behavior is similar to Conway's description of ideal transistors as switches [9]. Her design approach focused on how transistors are *used* and the accompanying expectations as a binary device, as opposed to giving details of its amplification performance as an analog device.

In what follows, the models of crypto operations, combined with the access-control logic, enable us to reason about systems using cryptographic-based authentication and authorization. In the following sections on symmetric key and asymmetric key encryption and decryption, cryptographic hash functions, and digital signatures, we describe the operation, how it is used, and the ideal behavior we model in HOL.

5.1 Symmetric-Key Cryptography

Figure 5.1 is a schematic of symmetric key encryption and decryption. Suppose Bob wishes to send a message to Alice that only he and Alice can read. Also suppose that Bob and Alice share the same secret key, which is also known as a *symmetric key*. Here are the steps that Bob and Alice take to communicate confidentially.

1. Bob *encrypts* his message in plaintext with the secret key k he shares with Alice. He forwards to encrypted message, i.e., the ciphertext, to Alice.

2. Alice uses symmetric key k to decrypt the ciphertext to retrieve the plaintext message.

Idealized Behavior

Symmetric-key cryptography is used with the following expectations: (1) the same key is the only means to decrypt what is encrypted, (2) if something useful and recognizable is decrypted, then it must mean that the decrypted text and the decryption key are identical to the original text and encryption key, and (3) using anything other than the original encryption key to decrypt will result in an unusable result. We capture these expectations semi-formally by the following statements.

1. Whatever is encrypted with key k is retrieved unchanged by decrypting with the same key k.

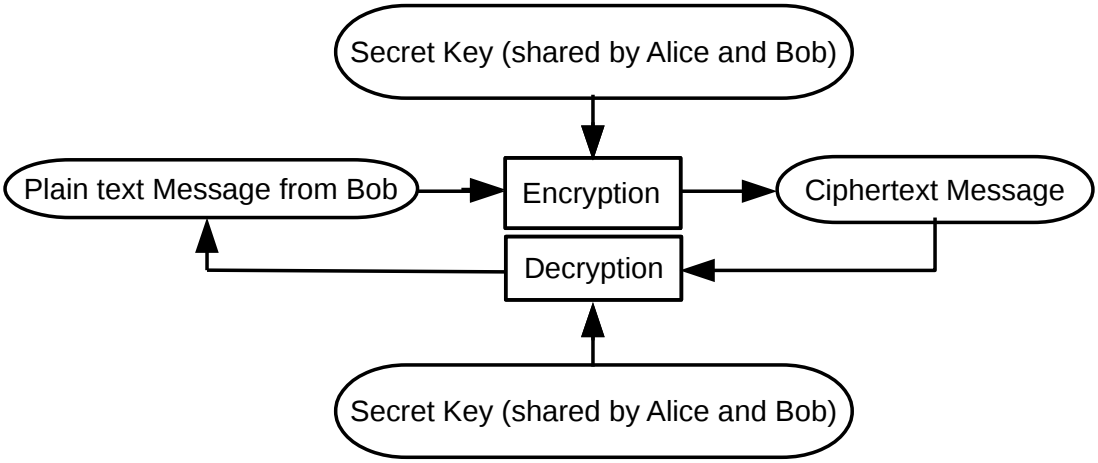
2. If key k_1 encrypted any plaintext, and key k_2 decrypted the resulting ciphertext and retrieved the original text, then $k_1 = k_2$.

3. If plaintext is encrypted with key k_1, decrypted with key k_2, and nothing useful results, then $k_1 \neq k_2$.

4. If nothing useful is encrypted using any key, then nothing useful is decrypted using any key.

Modeling Idealized Behavior in HOL

Figure 5.1 Symmetric-Key Encryption and Decryption



Adding "Nothing Useful" as a Value One aspect we must model is the notion of "nothing useful" as a value or result. To do this in a general fashion, we use *option* theory in HOL. Figure 5.2 shows the type definition of *option* and the properties of *option* types in HOL in the theorem *option_CLAUSES*.

The *option* type is polymorphic. *option* types are created from other types using the type constructor *SOME*. For example, when *SOME* is applied to the natural number 1, i.e., *SOME 1*, the resulting value is of type *num option*. The *num option* type has all the values of *SOME n*, where *n* is a natural number in HOL, with one added value: *NONE*. We use *NONE* when we want to return a value other than a natural number, e.g., in the case where we return a result of dividing by zero.

In the case of modeling encryption and decryption, we use *option* types to add the value *NONE* to whatever we are encrypting or decrypting. Doing so allows us to handle cases such as what value to return if the wrong key is used to decrypt an encrypted message.

Finally, the accessor function *THE* is used to retrieve the value to which *SOME* is applied. For example, $THE(SOME\ x) = x$, as shown in *option_CLAUSES*.

Symmetric Keys, Encryption, Decryption, and their Properties Figure 5.3 shows the definitions and properties of symmetric-key encryption and decryption. The following is a list of key definitions and properties.

- Symmetric keys are modeled by the algebraic type *symKey*. The type constructor is *sym*. For example, `sym 1234` is a symmetric key. Abstractly, `sym 1234` is the symmetric key which is identified by number 1234.

- Two symmetric keys are identical if they have the same number to which *sym* is applied. This is shown in theorem *symKey_one_one*.

- Symmetrically encrypted messages are modeled by the algebraic type *symMsg*, whose type constructor is *Es*. Symmetrically encrypted messages have two arguments: (1) a `symKey`, and (2) a `'message option`. For example, `Es (sym 1234) (SOME "This is a string")` is a symmetrically encrypted message using: (1) the symmetric key `sym 1234`, and (2) the *string option* value `SOME "This is a string"`. Abstractly, the type constructor *Es* stands for any symmetric-key encryption algorithm, e.g., DES or AES.

- Two *symMsg* values are identical if their corresponding components are identical. This is shown in theorem *symMsg_one_one*.

- Symmetric-key decryption of *symMsgs* is defined by *deciphS_def*. If the same *symKey* is used to decipher an encrypted *SOME x*, then *SOME x* is returned. Otherwise, *NONE* is returned. If nothing useful is encrypted, then nothing useful is decrypted. Abstractly, *deciphS* represents any symmetric key decryption algorithm.

Figure 5.2 Option Theory in HOL

```
option = NONE | SOME 'a

[option_CLAUSES]

⊢ (∀x y. (SOME x = SOME y) ⟺ (x = y)) ∧
   (∀x. THE (SOME x) = x) ∧ (∀x. NONE ≠ SOME x) ∧
   (∀x. SOME x ≠ NONE) ∧ (∀x. IS_SOME (SOME x) ⟺ T) ∧
   (IS_SOME NONE ⟺ F) ∧ (∀x. IS_NONE x ⟺ (x = NONE)) ∧
   (∀x. ¬IS_SOME x ⟺ (x = NONE)) ∧
   (∀x. IS_SOME x ⇒ (SOME (THE x) = x)) ∧
   (∀x. option_CASE x NONE SOME = x) ∧
   (∀x. option_CASE x x SOME = x) ∧
   (∀x. IS_NONE x ⇒ (option_CASE x e f = e)) ∧
   (∀x. IS_SOME x ⇒ (option_CASE x e f = f (THE x))) ∧
   (∀x. IS_SOME x ⇒ (option_CASE x e SOME = x)) ∧
   (∀v f. option_CASE NONE v f = v) ∧
   (∀x v f. option_CASE (SOME x) v f = f x) ∧
   (∀f x. OPTION_MAP f (SOME x) = SOME (f x)) ∧
   (∀f. OPTION_MAP f NONE = NONE) ∧ (OPTION_JOIN NONE = NONE) ∧
  ∀x. OPTION_JOIN (SOME x) = x
```

- Finally, *deciphS_clauses* is the HOL theorem that shows our type definitions for keys and encryption, coupled with our definition of decryption, has the properties we expect: (1) the same key when used for encryption and decryption returns the original message, (2) if the original message was retrieved, identical keys were used, (3) if a different key is used to decrypt ciphertext, then nothing useful is returned, and (4) garbage in and garbage out holds true.

5.2 Cryptographic Hash Functions

Cryptographic hash functions are used to map inputs of any size into a fixed number of bits. Cryptographic hash functions are one-way functions, (1) the output is easy to compute from the input, and (2) it is computationally infeasible to determine an input when given only a hash value. Hash values are also known as *digests*.

Figure 5.4 shows the type definition of *digest* and their properties. The following describes the type definition and its properties.

- Digests or hashes are modeled by the algebraic type *digest*. The type constructor is *hash* and is meant to represent any hash algorithm, e.g., SHA1 and SHA2. Notice that the *hash* is applied to polymorphic arguments of type `'message option`, e.g., `hash (SOME "A string message")`.

- The key property of *ideal* digests is they are one-to-one, as shown by the theorem *digest_one_one*. In reality, hashes cannot be one-to-one due to their fixed-length output. Modeling digests in this way is analogous to abstracting the electrical behavior of transistors as amplifiers away and idealizing them as perfect switches.

5.3 Asymmetric-Key Cryptography

Figure 5.5 is a schematic of asymmetric key encryption and decryption. The asymmetric nature of asymmetric key, or public-key cryptography, is two different keys are used instead of the same key. One key, known as a *public key*, may be freely disclosed. The other key, known as a *private key*, must be *known only by one principal*.

Suppose Alice wishes to send a message to Bob that only Bob can read. Alice encrypts the message to Bob using his public key K_{Bob}. Only Bob, who alone possesses the private key K_{Bob}^{-1}, is able to decrypt the message encrypted with his public key K_{Bob}.

Figure 5.3 Definitions and Properties of Symmetric Encryption and Decryption

```
symKey = sym num

[symKey_one_one]

⊢ ∀a a'. (sym a = sym a') ⟺ (a = a')

symMsg = Es symKey ('message option)

[symMsg_one_one]

⊢ ∀a0 a1 a0' a1'.
    (Es a0 a1 = Es a0' a1') ⟺ (a0 = a0') ∧ (a1 = a1')

[deciphS_def]

⊢ (deciphS k1 (Es k2 (SOME x)) =
   if k1 = k2 then SOME x else NONE) ∧
  (deciphS k1 (Es k2 NONE) = NONE)

[deciphS_clauses]

⊢ (∀k text. deciphS k (Es k (SOME text)) = SOME text) ∧
  (∀k1 k2 text.
     (deciphS k1 (Es k2 (SOME text)) = SOME text) ⟺
     (k1 = k2)) ∧
  (∀k1 k2 text.
     (deciphS k1 (Es k2 (SOME text)) = NONE) ⟺ k1 ≠ k2) ∧
  ∀k1 k2. deciphS k1 (Es k2 NONE) = NONE

[deciphS_one_one]

⊢ (∀k1 k2 text1 text2.
     (deciphS k1 (Es k2 (SOME text2)) = SOME text1) ⟺
     (k1 = k2) ∧ (text1 = text2)) ∧
  ∀enMsg text key.
    (deciphS key enMsg = SOME text) ⟺
    (enMsg = Es key (SOME text))
```

Asymmetric-key cryptography is used with the following expectations: (1) plaintext that is encrypted with a private key and can be retrieved only with the corresponding public key, (2) plaintext that is encrypted with a public key can be retrieved only with the corresponding private key, (3) if plaintext was retrieved that was encrypted with a private key, then the corresponding public key was was used to decrypt the ciphertext, (4) if plaintext was retrieved that was encrypted with a public key, then the corresponding private key was used to decrypt the ciphertext, and (5) nothing useful results if decryption uses anything but the corresponding public or private key used in encryption.

Figure 5.6 shows the type definitions for asymmetric keys *pKey*, i.e., public and private keys, and asymmetrically encrypted messages *asymMsg*. Figure 5.6 also shows properties of *pKey* and *asymMsg*.

- The type *pKey* has two forms, *pubK P* and *privK P*, public and private, respectively. Asymmetric keys are polymorphic and intended to be associated with principals *P* with variable type *'princ*.
- The private and public keys of any principal are not the same.
- Public and private keys are the same if they have the same parameters.
- The type *asymMsg* represents asymmetrically encrypted messages. The parameters of type constructor *Ea* are a *pKey* and a *'message option*. Abstractly, the type constructor *Ea* stands for any asymmetric-key algorithm, e.g., RSA.

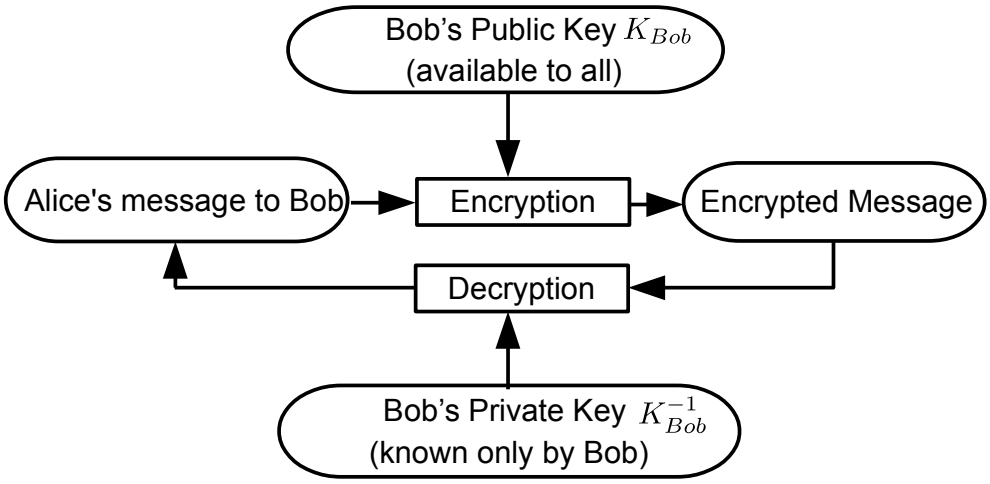
Figure 5.4 Definition of Digests and their Properties

```
digest = hash ('message option)

[digest_one_one]

⊢ ∀a a'. (hash a = hash a') ⟺ (a = a')
```

Figure 5.5 Asymmetric-Key Encryption and Decryption



- Two *asymMsgs* are the same if they have the same *pKey* and *'message option* values.

Figure 5.7 shows the definition and properties of *deciphP*, which models the decryption of asymmetrically encrypted messages. Similar to symmetric-key encryption, to retrieve the plaintext *SOME x* requires use of the correct key, in this case *privK P* if the message was encrypted using *pubK P*, or *pubK P* if the message was encrypted with *privK P*. As before, garbage in produces garbage out.

The properties of *deciphP* are shown in Figures 5.7 and 5.8 by theorems *deciphP_clauses* and *deciphP_one_one*. Together, they show the circumstances under which the original plaintext is decrypted, when nothing useful is decrypted, and the conditions that ensure that the expected keys and plaintext messages were in fact, used.

5.4 Digital Signatures

Digitally signed messages are often a combination of cryptographic hashes of messages encrypted using the *private key* of the sender. This is shown in Figure 5.9, which depicts signature generation as the following sequence of operations:

1. A message is hashed, then
2. the message hash is encrypted using the private key of the sender.

The intuition behind signatures is this: (1) the cryptographic hash is a unique pointer to the message (and potentially much smaller than the message), and (2) encrypting using the sender's private key (which is reversible by the sender's public key) is a unique pointer to the sender.

Figure 5.10 shows how decrypted messages are checked for integrity using digital signatures. The top-most sequence from left to right shows how the decrypted hash value is retrieved from the received digital signature. The digital signature is decrypted using the sender's public key to retrieve the hash or digest of the original message. The retrieved hash is compared to the hash of the decrypted message. If the two hash values are the same, then the received message is judged to have arrived unchanged from the original.

Figure 5.6 Definitions and Properties of Asymmetric Keys and Messages

pKey = pubK 'princ | privK 'princ

[pKey_distinct_clauses]

⊢ (∀a' a. pubK a ≠ privK a') ∧ ∀a' a. privK a' ≠ pubK a

[pKey_one_one]

⊢ (∀a a'. (pubK a = pubK a') ⟺ (a = a')) ∧
  ∀a a'. (privK a = privK a') ⟺ (a = a')

asymMsg = Ea ('princ pKey) ('message option)

[asymMsg_one_one]

⊢ ∀a_0 a_1 a'_0 a'_1.
    (Ea a_0 a_1 = Ea a'_0 a'_1) ⟺ (a_0 = a'_0) ∧ (a_1 = a'_1)

Figure 5.11 shows the function definitions in HOL of *sign* and *signVerify*. *sign* takes as inputs a *pKey* and a digest and returns an asymmetrically encrypted digest using the asymmetric *pKey*. *signVerify* takes as input a *pKey*, digital signature, and a received message and compares the decrypted hash in the signature with the hash of the received message. The properties of *signVerify* and *sign* are in theorems *signVerifyOK* and *signVerify_one_one*.

- *signVerify* is always true for signatures generated as shown in Figure 5.9.
- *signVerify* and *sign* combine to have the desired properties that the plaintext must match and the corresponding keys must match.

Figure 5.7 Definitions and Properties of Asymmetric Decryption

[deciphP_def]

```
⊢ (deciphP key (Ea (privK P) (SOME x)) =
    if key = pubK P then SOME x else NONE) ∧
  (deciphP key (Ea (pubK P) (SOME x)) =
    if key = privK P then SOME x else NONE) ∧
  (deciphP k1 (Ea k2 NONE) = NONE)
```

[deciphP_clauses]

```
⊢ (∀P text.
     (deciphP (pubK P) (Ea (privK P) (SOME text)) =
      SOME text) ∧
     (deciphP (privK P) (Ea (pubK P) (SOME text)) =
      SOME text)) ∧
  (∀k P text.
     (deciphP k (Ea (privK P) (SOME text)) = SOME text) ⟺
     (k = pubK P)) ∧
  (∀k P text.
     (deciphP k (Ea (pubK P) (SOME text)) = SOME text) ⟺
     (k = privK P)) ∧
  (∀x k2 k1 P2 P1.
     (deciphP (pubK P1) (Ea (pubK P2) (SOME x)) = NONE) ∧
     (deciphP k1 (Ea k2 NONE) = NONE)) ∧
  ∀x P2 P1. deciphP (privK P1) (Ea (privK P2) (SOME x)) = NONE
```

Figure 5.8 One-to-One Properties of Asymmetric Decryption

[deciphP_one_one]

```
⊢ (∀P1 P2 text1 text2.
     (deciphP (pubK P1) (Ea (privK P2) (SOME text2)) =
      SOME text1) ⟺ (P1 = P2) ∧ (text1 = text2)) ∧
  (∀P1 P2 text1 text2.
     (deciphP (privK P1) (Ea (pubK P2) (SOME text2)) =
      SOME text1) ⟺ (P1 = P2) ∧ (text1 = text2)) ∧
  (∀p c P msg.
     (deciphP (pubK P) (Ea p c) = SOME msg) ⟺
     (p = privK P) ∧ (c = SOME msg)) ∧
  (∀enMsg P msg.
     (deciphP (pubK P) enMsg = SOME msg) ⟺
     (enMsg = Ea (privK P) (SOME msg))) ∧
  (∀p c P msg.
     (deciphP (privK P) (Ea p c) = SOME msg) ⟺
     (p = pubK P) ∧ (c = SOME msg)) ∧
  ∀enMsg P msg.
    (deciphP (privK P) enMsg = SOME msg) ⟺
    (enMsg = Ea (pubK P) (SOME msg))
```

Figure 5.9 Digital Signature Generation

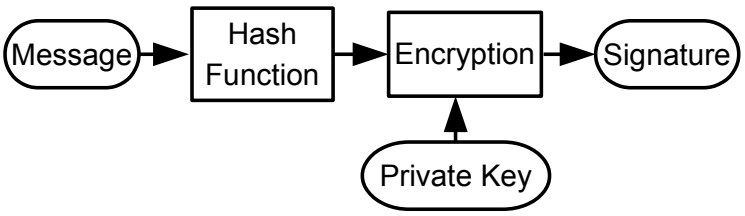


Figure 5.10 Digital Signature Verification

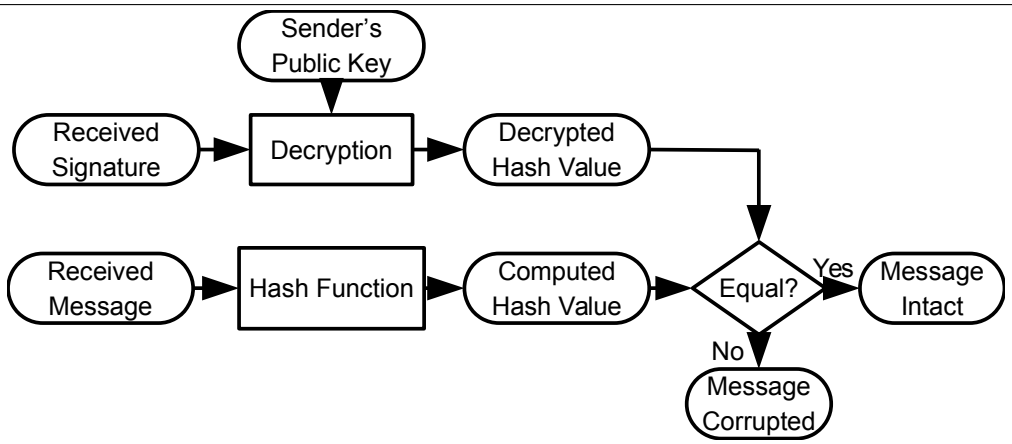


Figure 5.11 Digital Signature Generation, Verification, and Their Properties

```
[sign_def]

⊢ ∀pubKey dgst. sign pubKey dgst = Ea pubKey (SOME dgst)

[signVerify_def]

⊢ ∀pubKey signature msgContents.
    signVerify pubKey signature msgContents ⟺
    (SOME (hash msgContents) = deciphP pubKey signature)

[signVerifyOK]

⊢ ∀P msg.
    signVerify (pubK P) (sign (privK P) (hash (SOME msg)))
      (SOME msg)

[signVerify_one_one]

⊢ (∀P m1 m2.
     signVerify (pubK P) (Ea (privK P) (SOME (hash (SOME m1))))
       (SOME m2) ⟺ (m1 = m2)) ∧
  (∀signature P text.
     signVerify (pubK P) signature (SOME text) ⟺
     (signature = sign (privK P) (hash (SOME text)))) ∧
  ∀text2 text1 P2 P1.
    signVerify (pubK P1) (sign (privK P2) (hash (SOME text2)))
      (SOME text1) ⟺ (P1 = P2) ∧ (text1 = text2)
```

Chapter 6

Adding Security to State Machines

In this section, we use the infrastructure we have described in previous sections to add authentication and authorization to the description of state machines. Traditionally, this authentication and authorization was a function of virtual machine monitors (VMMs) or hypervisors. Our approach is to combine VMM functions into the description of state machines. We call these machines *secure state-machines* (SSMs).

At this point, we now have the following logical infrastructure:

1. An access-control logic and a C2 calculus in the form of inference rules implemented and verified as sound within the HOL theorem prover.
2. Algebraic models in HOL of cryptographic operations including symmetric and asymmetric encryption and decryption, cryptographic hashes, and digital signature generation and verification.
3. Parameterized state machines of arbitrary size described in HOL using labeled transition relations defined inductively in HOL.
4. Definitions and access-control logic interpretations of thermostat messages and certificates implemented in HOL.

Using the above infrastructure, we combine the above elements to extend the parameterized state-machine description in Chapter 3 to account for authentication and authorization. We do so at two levels.

1. State-machine transition behavior at a purely logical level where inputs and the security context are described in the access-control logic, and
2. State-machine transition behavior at a concrete level using (1) message and certificate data structures, and (2) interpretations in the access-control logic of messages and certificates.

There are many policies that define secure behavior, e.g., the classic military confidentiality policies of Bell and La Padula [6] and [5], the integrity policies of Biba [7], and role-based access control [10] and [13]. For illustrative purposes, the security policy we use is based on Popek and Goldberg's virtualization policies [12]. We chose virtualization because it lends itself to state-machine descriptions and it supports specifications where authorization and authentication are parameters.

The high-level policy followed by secure state-machines (SSMs) is as follows.

1. If an input to the machine fails to pass the supplied integrity check used by the machine, the input is discarded.
2. Inputs that are authenticated and deemed intact, are checked for authorization within the context of a state-interpretation function and list of certificates. An example of a security-interpretation of a state is when a *mode bit* is used to indicate if the machine is operating in privileged mode or user mode. An example of a certificate used for authorization is a *ticket* granting permission to access or use an object or service.
   (a) Authorized commands are *executed*.
   (b) Unauthorized commands are *trapped*.
3. Within the context of a specific application, commands are divided into two groups:
   (a) *Security-sensitive* commands, i.e., commands that if misused, compromise the integrity or confidentiality of operations, e.g., compromising process isolation, or

Figure 6.1 Definitions and Properties of Instruction and Transition Types

```
inst = CMD 'command | TRAP

[inst_distinct_clauses]

⊢ (∀a. CMD a ≠ TRAP) ∧ ∀a. TRAP ≠ CMD a

trType = discard | trap 'inst | exec 'inst

[trType_distinct_clauses]

⊢ (∀a. discard ≠ trap a) ∧ (∀a. discard ≠ exec a) ∧
   (∀a' a. trap a ≠ exec a') ∧ (∀a. trap a ≠ discard) ∧
   (∀a. exec a ≠ discard) ∧ ∀a' a. exec a' ≠ trap a
```

(b) *Innocuous* commands, i.e., commands that do not compromise integrity or confidentiality.

4. In keeping with the requirements for virtualizability as defined in [12], all security-sensitive commands are *privileged commands*, i.e., executable only by authorized principals. Attempts by unauthorized principals to execute privileged commands are trapped.

In keeping with making our SSM theories as reusable as possible, we fully parameterize them in terms of:

- authentication functions,
- authorization context given by lists of certificates and credentials, which have meaning in the access-control logic,
- functions for defining the meaning of inputs, certificates, and states, in the access-control logic,
- next-state functions,
- output functions, and
- type variables for inputs, outputs, and states in support of polymorphism.

We develop two levels of SSM description:

1. a high-level logical description relying on access-control logic formulas for inputs and certificates, and
2. a lower-level description using type variables and interpretation functions for inputs, states, and certificates.

This lower-level description is a refinement of the high-level description of behavior.

6.1 Instructions and Transition Types

Figure 6.1 shows the definition and properties of SSM instructions *inst* and state-transition types *trType*. The *inst* type is polymorphic, and is constructed with the type variable *'command* and the type constructor *CMD*. One additional instruction, *TRAP*, is added to all the commands that are in *'command*. The theorems *inst_distinct_clauses* and *trType_distinct_clauses* are the usual theorems stating that each form of *inst* or *trType* is distinct from the other.

There are two points regarding *inst*.

1. The purpose of *inst* is to add *TRAP* to the to the set of commands. Doing so facilitates writing policies in the access-control logic specifying when *TRAPs* should occur.
2. We can achieve the same effect by using *option* types, i.e., using *SOME* and *NONE*. To enhance readability, we use *CMD* and *TRAP* instead.

Figure 6.2 Definition and Properties of High-Level SSM Configurations

```
configuration =
    CFG (('command inst, 'principal, 'd, 'e) Form -> bool)
        ('state -> ('command inst, 'principal, 'd, 'e) Form)
        (('command inst, 'principal, 'd, 'e) Form list)
        (('command inst, 'principal, 'd, 'e) Form list) 'state
        ('output list)
```

[configuration_11]

$$\vdash \forall a_0\ a_1\ a_2\ a_3\ a_4\ a_5\ a'_0\ a'_1\ a'_2\ a'_3\ a'_4\ a'_5.$$
$$(\texttt{CFG}\ a_0\ a_1\ a_2\ a_3\ a_4\ a_5 = \texttt{CFG}\ a'_0\ a'_1\ a'_2\ a'_3\ a'_4\ a'_5) \iff$$
$$(a_0 = a'_0) \wedge (a_1 = a'_1) \wedge (a_2 = a'_2) \wedge (a_3 = a'_3) \wedge$$
$$(a_4 = a'_4) \wedge (a_5 = a'_5)$$

Figure 6.3 satList Definition and Properties

[satList_def]

$$\vdash \forall M\ Oi\ Os\ formList.$$
$$(M, Oi, Os)\ \texttt{satList}\ formList \iff$$
$$\texttt{FOLDR}\ (\lambda x\ y.\ x \wedge y)\ \texttt{T}\ (\texttt{MAP}\ (\lambda f.\ (M, Oi, Os)\ \texttt{sat}\ f)\ formList)$$

[satList_nil]

$$\vdash (M, Oi, Os)\ \texttt{satList}\ []$$

[satList_CONS]

$$\vdash \forall h\ t\ M\ Oi\ Os.$$
$$(M, Oi, Os)\ \texttt{satList}\ (h::t) \iff$$
$$(M, Oi, Os)\ \texttt{sat}\ h \wedge (M, Oi, Os)\ \texttt{satList}\ t$$

[satList_conj]

$$\vdash \forall l_1\ l_2\ M\ Oi\ Os.$$
$$(M, Oi, Os)\ \texttt{satList}\ l_1 \wedge (M, Oi, Os)\ \texttt{satList}\ l_2 \iff$$
$$(M, Oi, Os)\ \texttt{satList}\ (l_1\ \texttt{++}\ l_2)$$

6.2 High-Level Secure State-Machine Description

Configurations Defined Figure 6.2 shows the definition of high-level SSM *configurations* and their properties. *Configurations* have six components.

1. An authentication function with type `('command inst, 'principal, 'd, 'e)Form -> bool` that returns *true* or *false* when applied to inputs expressed as access-control logic formulas. This function determines whether or not commands originate from known and approved sources.

2. A state interpretation function with type `'state -> ('command inst, 'principal, 'd, 'e)Form` that maps a state into an access-control logic formula. The interpretation function and state are part of the security context informing the decision on whether or not an authenticated request is authorized.

3. A list of access-control logic formulas `('command inst, 'principal, 'd, 'e)Form list` that represent the security context, with security interpretation of the current state, in which authenticated requests are authorized or not. The list elements correspond to the meaning of certification, polices, trust assumptions, and authorizations in the access-control logic.

Figure 6.4 Definition of Configuration Interpretation Function

[CFGInterpret_def]

```
⊢ CFGInterpret (M,Oi,Os)
      (CFG inputTest stateInterp context (x::ins) state
         outStream) ⟺
    (M,Oi,Os) satList context ∧ (M,Oi,Os) sat x ∧
    (M,Oi,Os) sat stateInterp state
```

Figure 6.5 Discard Command Rule for Transition Relation TR

[TR_discard_cmd_rule]

```
⊢ TR (M,Oi,Os) discard
      (CFG inputTest stateInterp certs (x::ins) s outs)
      (CFG inputTest stateInterp certs ins (NS s discard)
         (Out s discard::outs)) ⟺ ¬inputTest x
```

4. An input stream of access-control logic formulas `('command inst, 'principal, 'd, 'e)Form list`.
5. The current state `'state`.
6. An output stream `'output list`.

The theorem *configuration_11* is the typical property stating that two *configurations* are identical if and only if all their components are identical.

Semantics of Lists of Access-Control Logic Formulas Defined To assist in the interpretation of *configurations*, we define the function *satList*, whose purpose is to give meaning to a list of access-control logic formulas, e.g., $[f_1; f_2; \cdots; f_n]$. Figure 6.3 defines *satList* and its properties. The net effect of the *satList* definition and theorems is *satList* applied to a Kripke structure M, partial orders O_i and O_s, and a list of access-control logic formulas $[f_1; f_2; \cdots; f_n]$, is that *satList* is the *and-reduction* of (M, O_i, O_s) *sat* mapped over each formula f_i. For example,

$$(M, O_i, O_s)\ satList\ [f_1; f_2; \cdots; f_n] = (M, O_i, O_s)\ sat\ f_1 \wedge \cdots \wedge (M, O_i, O_s)\ sat\ f_n$$

Interpretation of Configurations Defined Figure 6.4 shows the definition *CFGInterpret_def*, which defines the meaning of *configurations* in the access-control logic. Simply put, the security interpretation of a configuration is the conjunction of formulas (M, O_i, O_s) *sat* f_i, where f_i corresponds to the formulas in the list *context*, the meaning of input x, and the interpretation of *state*.

Configuration Transition Relation TR Defined and Its Properties We define inductively the transition relation *TR* on *configurations* using the same techniques as shown in Example 3.0.2. This time, we account for the security interpretation of *configurations*.

Appendix C.1 gives the HOL source code for defining TR. Appendix C.2 shows the three defining properties of *TR* in HOL resulting from the inductive definition. These properties are *TR_rules*, *TR_ind*, and *TR_cases*, which give the transition rules, induction property, and cases theorem, respectively.

Looking at *TR_rules*, we see there are three clauses, one each for the three *trTypes* labeling the transition relation *TR (M, Oi, Os)*:

1. *TR (M, Oi, Os) (exec (CMD cmd))*: the rule specifying when a command *cmd* is executed. The conditions are:
   (a) the input `P says prop (CMD cmd)` must be authenticated by *inputTest*, and
   (b) the security interpretation of the current configuration is given by *CFGInterpret*.

Figure 6.6 Execute Command Rule for Transition Relation TR

```
[TR_exec_cmd_rule]

⊢ ∀inputTest certs stateInterp P cmd ins s outs.
    (∀M Oi Os.
       CFGInterpret (M,Oi,Os)
         (CFG inputTest stateInterp certs
            (P says prop (CMD cmd)::ins) s outs) ⇒
       (M,Oi,Os) sat prop (CMD cmd)) ⇒
  ∀NS Out M Oi Os.
    TR (M,Oi,Os) (exec (CMD cmd))
      (CFG inputTest stateInterp certs
         (P says prop (CMD cmd)::ins) s outs)
      (CFG inputTest stateInterp certs ins
         (NS s (exec (CMD cmd)))
         (Out s (exec (CMD cmd))::outs)) ⟺
    inputTest (P says prop (CMD cmd)) ∧
    CFGInterpret (M,Oi,Os)
      (CFG inputTest stateInterp certs
         (P says prop (CMD cmd)::ins) s outs) ∧
    (M,Oi,Os) sat prop (CMD cmd)
```

2. *TR (M, Oi, Os) (trap (CMD cmd))*: the rule specifying when a command *cmd* is trapped. The conditions are:
   (a) the input `P says prop (CMD cmd)` must be authenticated by *inputTest*, and
   (b) the security interpretation of the current configuration is given by *CFGInterpret*.
3. *TR (M, Oi, Os) discard*: the rule specifying when an input *x* is discarded. The rule states when *x* fails to be authenticated by *inputTest*, *x* is discarded from the input stream.

Based on the definitions of *TR*, *satList*, and *CFGInterpret*, and their properties, we can prove three equality properties related to each of the transition types *trType*. The following three equality rules are parameterizable, convenient, and essential for easily certifying the security properties of devices such as the networked thermostat. The equality theorems are:

1. *TR_discard_cmd_rule* as shown in Figure 6.5. It states that a *discard* transition occurs for an input *x* if and only if *x* fails to be authenticated, i.e., ¬*inputTest x* is true.
2. *TR_exec_cmd_rule* as shown in Figure 6.6. It states that if `(M, Oi, Os) sat prop (CMD cmd)` is justified, i.e., implied by the security interpretation of the current configuration, as specified by *CFGInterpret*, then *cmd* is executed if and only if (a) the input is authenticated, (b) *CFGInterpret* is the security interpretation, and (c) `(M, Oi, Os) sat prop (CMD cmd)` is true.
3. *TR_trap_cmd_rule* as shown in Figure 6.7. It states that if `(M, Oi, Os) sat prop TRAP` is justified, i.e., implied by the security interpretation of the current configuration, as specified by *CFGInterpret*, then *cmd* is trapped if and only if (a) the input is authenticated, (b) *CFGInterpret* is the security interpretation, and (c) `(M, Oi, Os) sat prop TRAP` is true.

Note that in the above three theorems, the following functions and types are parameterized, making the theorems applicable to state machines in general using the concepts of discarding, trapping, and executing commands. The specific parameters are:

1. *inputTest*: the authentication function,
2. *stateInterp*: the state interpretation function,
3. *certs*: the credentials, trust assumptions, delegations, and authorizations informing authorization decisions,

Figure 6.7 Trap Command Rule for Transition Relation TR

```
[TR_trap_cmd_rule]

⊢ ∀inputTest stateInterp certs P cmd ins s outs.
    (∀M Oi Os.
       CFGInterpret (M,Oi,Os)
         (CFG inputTest stateInterp certs
            (P says prop (CMD cmd)::ins) s outs) ⇒
       (M,Oi,Os) sat prop TRAP) ⇒
    ∀NS Out M Oi Os.
      TR (M,Oi,Os) (trap (CMD cmd))
        (CFG inputTest stateInterp certs
           (P says prop (CMD cmd)::ins) s outs)
        (CFG inputTest stateInterp certs ins
           (NS s (trap (CMD cmd)))
           (Out s (trap (CMD cmd))::outs)) ⟺
      inputTest (P says prop (CMD cmd)) ∧
      CFGInterpret (M,Oi,Os)
        (CFG inputTest stateInterp certs
           (P says prop (CMD cmd)::ins) s outs) ∧
      (M,Oi,Os) sat prop TRAP
```

4. *commands*: commands are polymorphic,
5. *states*: states are polymorphic,
6. *outputs*: outputs are polymorphic,
7. *NS*: the next-state function, and
8. *Out*: the output function.

The three theorems in Figures 6.5, 6.6, and 6.7, provide a parameterized framework at the *logic design level* of state machines. We use this framework in specific applications, such as the networked thermostat, by specifying each of the eight parameters listed above.

Where is assurance of security accounted for in these theorems?

1. In *TR_discard_cmd_rule* the authentication function *inputTest* eliminates all unauthenticated commands.
2. In *TR_exec_cmd_rule*, the condition

   ```
   ∀M Oi Os.
    CFGInterpret (M,Oi,Os)
     (CFG inputTest stateInterp certs (P says prop (CMD cmd)::ins) s outs) ⇒
    (M,Oi,Os) sat prop (CMD cmd)
   ```

   corresponds to a *derived inference rule* in the C2 calculus. In effect, the theorem states that if the above is proved to be a theorem in the C2 calculus, then the remaining if and only if clause of the theorem holds.
3. In *TR_trap_cmd_rule*, similar to *TR_exec_cmd_rule*, the condition

   ```
   ∀M Oi Os.
    CFGInterpret (M,Oi,Os)
    (CFG inputTest stateInterp certs (P says prop (CMD cmd)::ins) s outs) ⇒
   (M,Oi,Os) sat prop TRAP
   ```

   corresponds to a *derived inference rule* in the C2 calculus. In effect, the theorem states that if the above is proved to be a theorem in the C2 calculus, then the remaining if and only if clause of the theorem holds.

Figure 6.8 Definition and Properties of Refined SSM Configurations and Interpretation

```
configuration2 =
    CFG2 ('input -> ('command inst, 'principal, 'd, 'e) Form)
         ('cert -> ('command inst, 'principal, 'd, 'e) Form)
         (('command inst, 'principal, 'd, 'e) Form -> bool)
         ('cert list)
         ('state -> ('command inst, 'principal, 'd, 'e) Form)
         ('input list) 'state ('output list)

[CFG2Interpret_def]

⊢ CFG2Interpret (M,Oi,Os)
     (CFG2 inputInterpret certInterpret inputTest certs
        stateInterpret (x::ins) state outStream) ⟺
   (M,Oi,Os) satList MAP certInterpret certs ∧
   (M,Oi,Os) sat inputInterpret x ∧
   (M,Oi,Os) sat stateInterpret state
```

6.3 Secure State-Machines Using Message and Certificate Structures

The previous high-level state-machine descriptions relied on access-control logic formulas only. To illustrate how details such as message and certificate structures are introduced, we develop a SSM description using polymorphic messages and certificates, and corresponding interpretation functions. We show that the transition relations *TR* and *TR2* are logically equivalent when they are applied to their corresponding configurations.

Configurations and Their Interpretations Defined Figure 6.8 shows the type definition of *configuration2* and its interpretation function *CFG2Interpret*. The refined configuration *configuration2* has eight components.

1. An input interpretation function with type `'input -> ('command inst, 'principal, 'd,'e)Form`. This function gives meaning to inputs in the access-control logic.
2. A certificate interpretation function with type `'cert -> ('command inst, 'principal, 'd,'e)Form`. This function gives meaning to certificates in the access-control logic.
3. An authentication function with type `('command inst, 'principal, 'd, 'e)Form -> bool` that returns *true* or *false* when applied to inputs expressed as access-control logic formulas. This function determines whether or not commands originate from known and approved sources.
4. A list of certificates with type `'cert list` that represent the security context, with security interpretation of the current state, in which authenticated requests are authorized or not.
5. A state interpretation function with type `'state -> ('command inst, 'principal, 'd, 'e)Form` that maps a state into an access-control logic formula. The interpretation function and state are part of the security context informing the decision on whether or not an authenticated request is authorized.
6. An input stream of access-control logic formulas `'input list`.
7. The current state `'state`.
8. An output stream `'output list`.

Configuration Transition Relation TR2 Defined and Its Properties We define inductively the transition relation *TR2* in an analogous way to the definition of *TR*. Appendix D.1 gives the HOL source code for defining *TR2*. Appendix D.2 shows the three defining properties of *TR2* in HOL resulting from the inductive definition. These properties are *TR2_rules*, *TR2_ind*, and *TR2_cases*, which are the transition rules, induction property, and cases theorem, respectively.

Figure 6.9 Discard Command Rule for Transition Relation TR2

[TR2_discard_cmd_rule]

```
⊢ TR2 (M, Oi, Os) discard
      (CFG2 inputInterpret certInterpret inputTest certs
         stateInterpret (x::ins) state outStream)
      (CFG2 inputInterpret certInterpret inputTest certs
         stateInterpret ins (NS state discard)
         (Out state discard::outStream)) ⟺
   ¬inputTest (inputInterpret x)
```

Figure 6.10 Execute Command Rule for Transition Relation TR2

[TR2_exec_cmd_rule]

```
⊢ ∀inputInterpret certInterpret inputTest certs stateInterpret
      x cmd ins state outStream.
      (∀M Oi Os.
         CFG2Interpret (M, Oi, Os)
            (CFG2 inputInterpret certInterpret inputTest certs
               stateInterpret (x::ins) state outStream) ⇒
         (M, Oi, Os) sat prop (CMD cmd)) ⇒
      ∀NS Out M Oi Os.
        TR2 (M, Oi, Os) (exec (CMD cmd))
          (CFG2 inputInterpret certInterpret inputTest certs
             stateInterpret (x::ins) state outStream)
          (CFG2 inputInterpret certInterpret inputTest certs
             stateInterpret ins (NS state (exec (CMD cmd)))
             (Out state (exec (CMD cmd))::outStream)) ⟺
        inputTest (inputInterpret x) ∧
        CFG2Interpret (M, Oi, Os)
          (CFG2 inputInterpret certInterpret inputTest certs
             stateInterpret (x::ins) state outStream) ∧
        (M, Oi, Os) sat prop (CMD cmd)
```

Based on the defining properties of *TR2* and *CFG2Interpret*, similar to *TR*, we prove three equality properties for the three transition types, *discard*, *exec (CMD cmd)*, and *trap (CMD cmd)*.

Note that in the above three theorems, the following functions and types are parameterized, making the theorems applicable to state machines in general using the concepts of discarding, trapping, and executing commands. The specific parameters are:

1. *inputInterpret*: the input interpretation function
2. *certInterpret*: the interpretation function for certificates
3. *inputTest*: the authentication function,
4. *stateInterp*: the state interpretation function,
5. *certs*: the credentials, trust assumptions, delegations, and authorizations informing authorization decisions,
6. *commands*: commands are polymorphic,
7. *states*: states are polymorphic,
8. *outputs*: outputs are polymorphic,
9. *NS*: the next-state function, and

Figure 6.11 Trap Command Rule for Transition Relation TR2

```
[TR2_trap_cmd_rule]

⊢ ∀inputInterpret certInterpret inputTest certs stateInterpret
      x cmd ins state outStream.
     (∀M Oi Os.
        CFG2Interpret (M,Oi,Os)
          (CFG2 inputInterpret certInterpret inputTest certs
             stateInterpret (x::ins) state outStream) ⇒
        (M,Oi,Os) sat prop TRAP) ⇒
    ∀NS Out M Oi Os.
      TR2 (M,Oi,Os) (trap (CMD cmd))
        (CFG2 inputInterpret certInterpret inputTest certs
           stateInterpret (x::ins) state outStream)
        (CFG2 inputInterpret certInterpret inputTest certs
           stateInterpret ins (NS state (trap (CMD cmd)))
           (Out state (trap (CMD cmd))::outStream)) ⟺
      inputTest (inputInterpret x) ∧
      CFG2Interpret (M,Oi,Os)
        (CFG2 inputInterpret certInterpret inputTest certs
           stateInterpret (x::ins) state outStream) ∧
       (M,Oi,Os) sat prop TRAP
```

10. *Out*: the output function.

The three theorems in Figures 6.9, 6.10, and 6.11, provide a parameterized framework for state machines with specific formats for inputs and certificates. We use this framework in specific applications, such as the networked thermostat, by specifying each of the eight parameters listed above.

In exactly the same way for *TR*, assurance of security is accounted for in *TR2* as follows.

1. In *TR2_discard_cmd_rule* the authentication function *inputTest* eliminates all unauthenticated commands.

2. In *TR2_exec_cmd_rule*, the condition

   ```
   ∀M Oi Os.
    CFG2Interpret (M,Oi,Os)
     (CFG2 inputInterpret certInterpret inputTest certs
           stateInterpret (x::ins) state outStream) ⇒
    (M,Oi,Os) sat prop (CMD cmd)
   ```

   corresponds to a *derived inference rule* in the C2 calculus. In effect, the theorem states that if the above is proved to be a theorem in the C2 calculus, then the remaining if and only if clause of the theorem holds.

3. In *TR2_trap_cmd_rule*, similar to *TR_exec_cmd_rule*, the condition

   ```
   ∀M Oi Os.
    CFG2Interpret (M,Oi,Os)
     (CFG2 inputInterpret certInterpret inputTest certs
           stateInterpret (x::ins) state outStream) ⇒
    (M,Oi,Os) sat prop TRAP
   ```

   corresponds to a *derived inference rule* in the C2 calculus. In effect, the theorem states that if the above is proved to be a theorem in the C2 calculus, then the remaining if and only if clause of the theorem holds.

Chapter 7

A Networked Thermostat Certified Secure by Design

Based on the all of the previous sections, we develop a networked thermostat that is certified secure by design. We pick up where we left off in Section 1.2.2, which gave the top-level CONOPS of a networked thermostat. In the descriptions that follow, we start from a top-level CONOPS and end with two secure state-machine descriptions (SSM) of the thermostat. The first SSM is a high-level logical description. The second SSM is a refinement of the first.

The tasks we need to do are:

1. Enumerate all commands and segregate them into two classes: privileged and non-privileged.
2. Enumerate all principals and their associated privileges within the envisioned thermostat operating modes.
3. Enumerate all thermostat use cases.
4. Specify the certificates needed to support authentication and authorization for all the use cases.
5. Devise the top-level SSM description by specializing *configurations* with definitions of
   (a) an authentication function,
   (b) a set of certificates described as formulas in the access-control logic,
   (c) a type for thermostat states,
   (d) a state-interpretation function,
   (e) a next-state function, and
   (f) an output function.
6. Formally define what is meant by the term "security" by defining a *security property* that is preserved by all thermostat SSM descriptions. Prove that all SSM descriptions satisfy the defined security property.
7. Refine the top-level SSM description into a second more detailed SSM description by augmenting the top-level SSM description with definitions of
   (a) an input message datatype,
   (b) an input message interpretation function,
   (c) a certificate datatype, and
   (d) a certificate interpretation function.
8. Prove the top-level and refined SSM descriptions are equivalent.

7.1 Thermostat Commands: Privileged and Non-Privileged

Section 1.2.2, has a high-level description of thermostat commands, which we summarize as follows.

1. **Setting** the temperature *value*.
2. **Enabling** the *Utility* to exercise control over setting the temperature.
3. **Disabling** the *Utility* to exercise control over setting the temperature.

Figure 7.1 Thermostat Commands and Properties

```
privcmd = Set num | EU | DU

npriv = Status

command = PR privcmd | NP npriv

[privcmd_distinct_thm]

⊢ (∀a. Set a ≠ EU) ∧ (∀a. Set a ≠ DU) ∧ EU ≠ DU

[privcmd_nchotomy_thm]

⊢ ∀pp. (∃n. pp = Set n) ∨ (pp = EU) ∨ (pp = DU)

[set_privcmd_11]

⊢ ∀a a'. (Set a = Set a') ⟺ (a = a')

[npriv_nchotomy_thm]

⊢ ∀a. a = Status

[command_distinct_thm]

⊢ ∀a' a. PR a ≠ NP a'

[command_nchotomy_thm]

⊢ ∀cc. (∃p. cc = PR p) ∨ ∃n. cc = NP n

[set_command_11]

⊢ (∀a a'. (PR a = PR a') ⟺ (a = a')) ∧
   ∀a a'. (NP a = NP a') ⟺ (a = a')
```

4. Reporting the **Status** of the thermostat, which is displayed on the thermostat and sent to the *Server*.

Temperature setting, enabling, and disabling the *Utility's* ability to exercise control over the thermostat are viewed as security-sensitive commands, as they can change the temperature setting and operating mode of the thermostat. In contrast, the *Status* command is innocuous, i.e., not security-sensitive, because reporting the thermostat's temperature setting and operating mode changes nothing.

The above partitioning of commands into two types (sensitive and non-sensitive) and why, is vital to incorporating security into designs from the beginning. In the case of the thermostat, the underlying basis for declaring a command to be security sensitive or not is whether or not the command in question can change either the temperature setting or operating mode.

Figure 7.1 shows the definitions of thermostat commands as types in HOL and their properties. These definitions incorporate the distinctions between security-sensitive and innocuous commands. The *privcmd* type has three thermostat commands, each of which are security sensitive and require *Owner* level privileges to execute.

1. *Set num*, which sets the temperature setting to the number supplied,

2. *EU*, which enables the *Utility* to control the thermostat, and

3. *DU*, which disables the *Utility* from controlling the thermostat.

The type *npriv* has a single thermostat command *Status*, which is innocuous and does not require *Owner* level privileges to execute. The type *command* defines all the thermostat commands into a single type using the type constructor *PR* for privileged commands *privcmd*, and the type constructor *NP* for *npriv* commands.

Figure 7.2 Networked Thermostat Principals

```
keyPrinc = CA | Server | Utility num

principal =
     Role keyPrinc
   | Key (keyPrinc pKey)
   | Keyboard
   | Owner num
   | Account num num
```

Figure 7.1 has seven theorems describing the properties of commands. The *distinct* theorems state that each command is different than all the others in its type. The *nchotomy* theorems completely enumerate the values or forms of a member of the particular type can have. The *_11* theorems, e.g., *set_privcmd_11*, state (where applicable) that identical values have identical components.

7.2 Thermostat Principals and Their Privileges

Recall Figure 1.2 in Section 1.2.2, which shows a networked thermostat receiving commands from two sources: (1) a *Keyboard* directly connected to it, and (2) the *Server* using a network interface. The operating assumptions are (1) all commands received from the *Keyboard* are from the *Owner*, and (2) the *Server* is relaying commands from the *Owner* or a *Utility*. *Owners* and *Utilities* have a unique ID number, where ID numbers are modeled as natural numbers.

Principals

Figure 7.2 shows the type definitions of the principals that interact with the thermostat. The *keyPrinc* type defined principals that will have asymmetric cryptographic keys. These principals are

1. *CA*: the Certificate Authority issuing public-key certificates.
2. *Server*: the *Server* relaying messages from the *Owner* or *Utility* to the thermostat.
3. *Utility*: the *Utility* with a numerical identifier to distinguish among the various utilities.

The *principal* type has five kinds of principals.

1. Principals that are *keyPrincs*, e.g., `CA, Server,` or `Utility utilityID`.
2. Public keys of *keyPrincs*, e.g., `Key (pubK CA)`—the public-key of the certificate authority *CA*.
3. A *Keyboard* attached to a thermostat.
4. An *Owner* with a unique numerical identifier to distinguish a thermostat and its owner from all other thermostats.
5. An *Account* on the *Server* with two numerical identifiers, one corresponding to the owner and the second corresponding to a PIN or password.

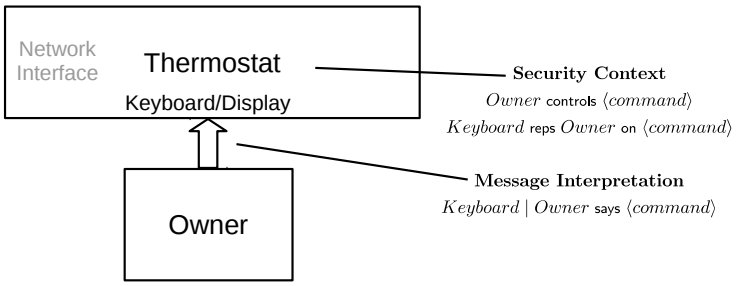
Privileges

Principals and their associated privileges are shown in Table 7.1. Any command involving *Owners* is authorized. *Utilities* are authorized on innocuous (non-security sensitive) commands. Utilities execute privileged commands only if the thermostat's operating mode is in a state that gives utilities authorization. All other listed principals have no authorization to execute any command, innocuous or otherwise.

Principal	Innocuous Commands	Privileged Commands
Owner	Yes	Yes
Keyboard | *Owner*	Yes	Yes
Server | *Owner*	Yes	Yes
Server | *Utility*	Yes	Yes, when *Utility* is *enabled* for control. No, otherwise.
Public keys	No	No
CA	No	No
Owner accounts	No	No

Table 7.1: Principals and Their Associated Privileges

Figure 7.3 Manual Operation



7.3 Thermostat Use Cases

Manual Operation

The thermostat is operated manually whenever the physical controls on the thermostat are used. The presumption is if the thermostat is operated manually then the *Owner* is behind the commands. This use case is illustrated, with the security context of the thermostat, in Figure 7.3.

When commands come from the *Keyboard*, the interpretation of what is received is *Keyboard* | *Owner* says $\langle command \rangle$. The thermostat's security context is:

1. The *Owner* has full authority over all commands, i.e., *Owner* controls $\langle command \rangle$.
2. The *Keyboard* is the *Owner's* delegate on $\langle command \rangle$. This is represented as *Keyboard* reps *Owner* on $\langle command \rangle$.

User Control Via the Server

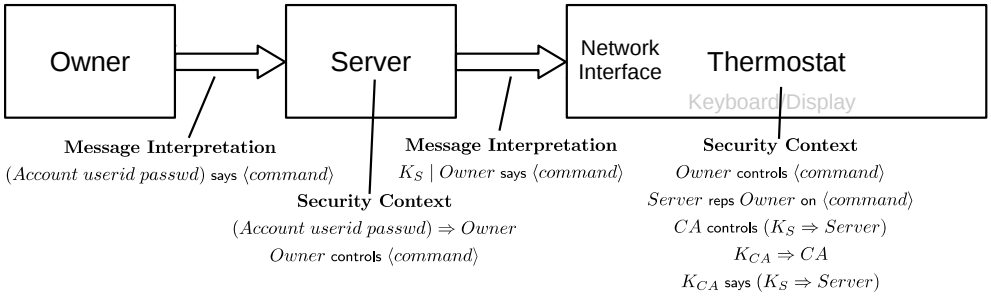
The thermostat is also controlled by the *Owner* through the *Owner's Account* on the *Server*. Figure 7.4 illustrates the messages and security context of the *Server* and thermostat. The *Server* and thermostat assume that the *Owner* has complete authority over all commands executed by the thermostat.

The *Server* identifies the *Owner* by the *Account userid passwd* associated with the *Owner*. After the *Server* authenticates the command from the *Owner*, it relays the command to the thermostat in a message that is cryptographically signed using its private key K_S^{-1}. If the cryptographically signed message passes the integrity check using the *Server's* public key K_S, then the message is interpreted to be K_S | *Owner* says $\langle command \rangle$ by the thermostat.

The thermostat's security context assumes:

1. The *Owner* has full authority over all commands, *Owner* controls $\langle command \rangle$.
2. The *Server* is the *Owner's* delegate on $\langle command \rangle$. This is *Server* reps *Owner* on $\langle command \rangle$.
3. The public key of the certificate authority CA is K_{CA}, i.e., $K_{CA} \Rightarrow CA$.
4. CA is trusted on public keys, i.e., CA controls $(K_S \Rightarrow Server)$.

Figure 7.4 Owner Control Via the Server



Utility Control Via the Server

When the *Utility* wishes to take control of the *Owner's* thermostat, e.g., to reduce air conditioning loads during peak power periods during the work day, the *Utility* will send the *Server* a command cryptographically signed by its private key K_U^{-1}. If the cryptographically signed message passes the integrity check using the *Utility's* public key K_U, then the message is interpreted to be $K_U \mid Owner$ says $\langle priv\ cmd \rangle$.

The *Server* has the content to authenticate the *Utility's* message by verifying the cryptographic signature. The part of the security context of the *Server* dealing with *Utility* authentication is:

1. *CA* controls $(K_U \Rightarrow Utility)$, i.e. the *Server* trusts *CA* on public keys.
2. K_{CA} says $(K_U \Rightarrow Utility)$. This is the public-key certificate for K_U cryptographically signed by K_{CA}.
3. $K_{CA} \Rightarrow CA$. This is a root trust assumption of the *Server* that K_{CA} is indeed *CA's* public key.

The remaining formulas in the *Server's* security context all deal with establishing the conditions under which the *Server* passes on the *Utility's* privileged command $\langle priv\ cmd \rangle$. Specifically, the following three formulas set the context for the *Owner* authorizing the *Server* to forward commands to the *Owner's* thermostat. The first formula states that the *Owner* has authority to authorize the Server to forward the request. The second formula is the actual authorization by *Account userid passwd*, the *Owner's* account. The third formula associates *Account userid passwd* with the *Owner*.

1. *Owner* controls $(Utility \mid Owner$ says $\langle priv\ cmd \rangle \supset Utility$ says $\langle priv\ cmd \rangle)$
2. $(Account\ userid\ passwd)$ says $(Utility \mid Owner$ says $\langle priv\ cmd \rangle \supset Utility$ says $\langle priv\ cmd \rangle)$
3. $(Account\ userid\ passwd) \Rightarrow Owner$

The next two sections discuss the security context of the thermostat. The first section shows the security context for authorizing the *Utility* to execute privileged commands, e.g., changing the temperature setting, on the thermostat. The second section shows the case when the thermostat has not authorized the *Utility* to execute privileged commands. If the *Utility* attempts to execute a privileged command, then it is trapped.

Both use cases share the same security context stating that the *Owner* has authority on privileged commands, the *Server* is the *Owner's* delegate, and the statements related to public-key certificates, the *CA's* authority, and the root trust assumption on the *CA's* public key. The last statement says that the *Server* is the *Utility's* delegate on privileged commands.

1. *Owner* controls $\langle priv\ cmd \rangle$
2. *Server* reps *Owner* on $\langle priv\ cmd \rangle$
3. *CA* controls $(K_S \Rightarrow Server)$
4. $K_{CA} \Rightarrow CA$
5. K_{CA} says $(K_S \Rightarrow Server)$
6. *Server* reps $Utility$ on $\langle priv\ cmd \rangle$

Figure 7.5 Utility Control Via the Server—Utility is Authorized

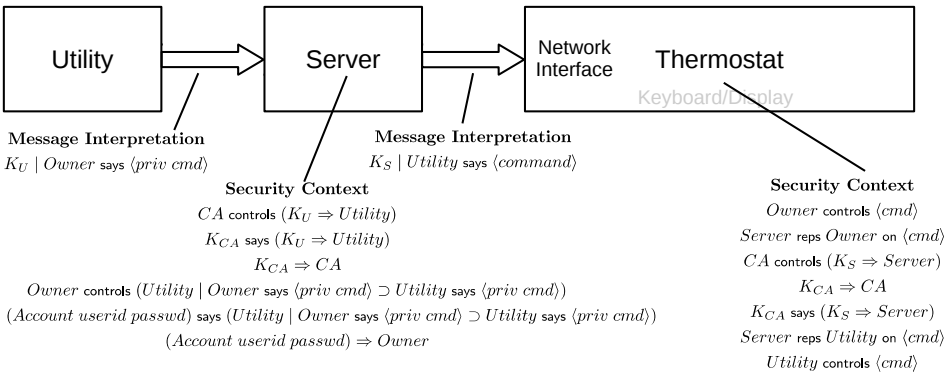
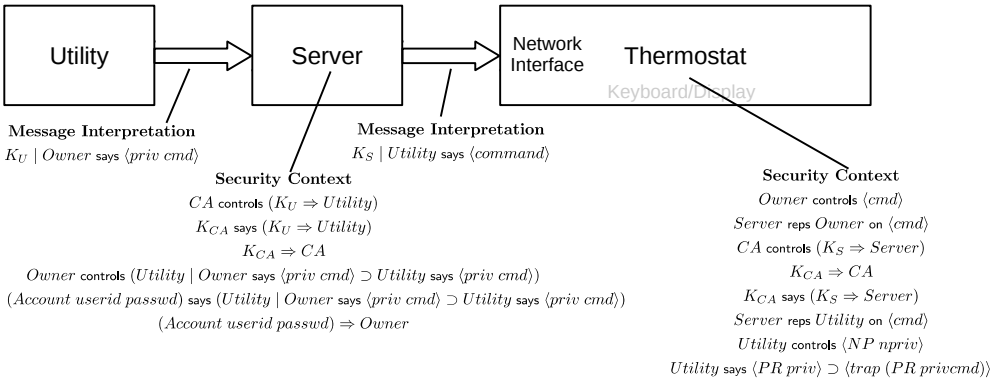


Figure 7.6 Utility Control Via the Server—Utility is Not Authorized



Utility Control is Authorized on All Commands Figure 7.5 illustrates the use case where the *Utility* is authorized by the *Owner* to exercise privileged commands, such as changing the temperature setting of the thermostat. The additional statement

$$Utility \text{ controls } \langle NP\ npriv \rangle$$
$$Utility \text{ controls } \langle PR\ privcmd \rangle$$

authorizes the *Utility* to execute all (privileged and non-privileged) commands.

Utility Control is Not Authorized on Privileged Commands Figure 7.6 illustrates the use case where the *Utility* is **not** authorized by the *Owner* to exercise privileged commands, such as changing the temperature setting of the thermostat, but is authorized to execute non-privileged (innocuous) commands. The additional statements

$$Utility \text{ controls } \langle NP\ npriv \rangle$$
$$Utility \text{ says } \langle PR\ privcmd \rangle \supset \langle trap\ (PR\ privcmd) \rangle$$

forces privileged commands issued by the *Utility* to be trapped.

7.4 Security Contexts for the Server and Thermostat

Server Security Context

The combined security context covering all the use cases is as follows. An explanation of the intent of each formula follows the formulas below.

1. $Owner$ controls $\langle cmd \rangle$
2. $(Account\ userid\ passwd) \Rightarrow Owner$
3. CA controls $(K_U \Rightarrow Utility)$
4. K_{CA} says $(K_U \Rightarrow Utility)$
5. $K_{CA} \Rightarrow CA$
6. $Owner$ controls $(Utility \mid Owner$ says $\langle cmd \rangle \supset Utility$ says $\langle cmd \rangle)$
7. $(Account\ userid\ passwd)$ says $(Utility \mid Owner$ says $\langle cmd \rangle \supset Utility$ says $\langle cmd \rangle)$

Formulas 1–2 Formula one states the *Owner's* authority to execute any command on her thermostat. Formula 2 states the association between the *Owner* and her account *Account userid passwd* on the *Server*.

Formulas 3–5 Formulas three through five deal with certificate authorities, root *CA* public keys, and public-key certificates. The third formula recognizes *CA's* is trusted on distributing the public key of the *Server*. The fourth formula corresponds to the public-key certificate of the *Server* digitally signed by *CA's* private key. The fifth formula is a root trust assumption stating that K_{CA} is *CA's* public key.

Formulas 6–7 The formulas six and seven state the *Owner's* authority and statement to authorize the *Server* to pass on commands from the *Utility* to the *Owner's* thermostat.

Thermostat Security Context

The thermostat has two mutually exclusive operating modes, (1) the *Utility* is authorized to execute privileged security-sensitive commands, which the thermostat will execute when received from the *Utility* relayed by the *Server*, or (2) the *Utility* is unauthorized on privileged commands and will trap any attempt by the *Utility* to execute a privileged command received from the *Utility* relayed by the *Server*. As a preview to our next Chapter, we handle mutually-exclusive operating modes by *changing thermostat configurations*. These mode or configuration changes switch security contexts and the commands to switch from one context to another are privileged and regarded as security sensitive. Such configuration changes are described by labeled transitions generally, e.g., high-level state machine descriptions, and by inductively-defined relations in HOL.

The common security context shared in both operating contexts is shown below. The intent of each formula follows the formulas below.

1. $Owner$ controls $\langle cmd \rangle$
2. $Keyboard$ reps $Owner$ on $\langle cmd \rangle$
3. $Server$ reps $Owner$ on $\langle cmd \rangle$
4. CA controls $(K_S \Rightarrow Server)$
5. $K_{CA} \Rightarrow CA$
6. K_{CA} says $(K_S \Rightarrow Server)$
7. $Server$ reps $Utility$ on $\langle NP\ npriv \rangle$
8. $Server$ reps $Utility$ on $\langle PR\ privcmd \rangle$
9. $Utility$ controls $\langle NP\ npriv \rangle$

Figure 7.7 Definition of high-level certificate list in HOL

```
[certs_def]

⊢ ∀ownerID utilityID cmd npriv privcmd.
    certs ownerID utilityID cmd npriv privcmd =
    [Name (Owner ownerID) controls prop (CMD cmd);
     reps (Name Keyboard) (Name (Owner ownerID))
       (prop (CMD cmd));
     reps (Name (Role Server)) (Name (Owner ownerID))
       (prop (CMD cmd));
     Name (Role CA) controls
     Name (Key (pubK Server)) speaks_for Name (Role Server);
     Name (Key (pubK CA)) speaks_for Name (Role CA);
     Name (Key (pubK CA)) says
     Name (Key (pubK Server)) speaks_for Name (Role Server);
     reps (Name (Role Server))
       (Name (Role (Utility utilityID)))
       (prop (CMD (NP npriv)));
     reps (Name (Role Server))
       (Name (Role (Utility utilityID)))
       (prop (CMD (PR privcmd)));
     Name (Role (Utility utilityID)) controls
     prop (CMD (NP npriv))]
```

Formulas 1–3 The first formula states the *Owner's* authority to execute any command $\langle cmd \rangle$. The second formula states the *Keyboard* is the *Owner's* delegate. In later refinements of the thermostat, we will interpret anything typed on the *Keyboard* as $Keyboard \mid Owner$. The third formula states that the *Server* is trusted to be the *Owner's* delegate when the *Server* quotes the *Owner*. **Note:** this points to a risk with networked devices—the devices must trust the integrity of their servers.

Formulas 4–6 The fourth, fifth, and sixth formulas deal with certificate authorities, root *CA* public keys, and public-key certificates. The fourth formula recognizes *CA's* is trusted on distributing the public key of the *Server*. The fifth formula corresponds to the public-key certificate of the *Server* digitally signed by *CA's* private key. The sixth formula is a root trust assumption stating that K_{CA} is *CA's* public key.

Formulas 7–9 The seventh and eighth formulas state that the *Server* is trusted to be the *Utility's* delegate when the *Server* quotes the *Utility* on both non-privileged and privileged commands. The ninth and final formula states that the *Utility* is authorized to execute non-privileged commands on the thermostat, e.g., query the status of the thermostat. **Note:** the thermostat is relying again upon the integrity of the *Server* to quote the correct originating principal behind a command. If the *Server* quotes the wrong principal, e.g., quotes the *Owner* instead of the *Utility*, then the thermostat potentially is duped into executing an unauthorized privileged instruction.

High-level certificates defined in HOL Figure 7.7 shows the definition of *certs* in HOL. The definition of *certs* is a list of access-control logic formulas in HOL corresponding to Formulas 1 through 9 above.

7.5 Top-Level Thermostat Secure State-Machine

The top-level thermostat SSM description is an instantiation of the high-level secure state-machine description in Section 6.2. The top-level thermostat SSM specializes the general high-level SSM with the following instantiations.

1. The type variable `'command` is instantiated with type *command*, as defined in Figure 7.1.
2. The type variable `'state` is instantiated with type *state* defined below.

Figure 7.8 HOL Type Definitions of mode and state

```
mode = enabled | disabled

state = State mode num
```

Figure 7.9 Thermostat State Interpretation Function

```
[thermoStateInterp_def]

⊢ (thermoStateInterp utilityID privcmd (State enabled temp) =
    Name (Role (Utility utilityID)) controls
    prop (CMD (PR privcmd))) ∧
   (thermoStateInterp utilityID privcmd (State disabled temp) =
    Name (Role (Utility utilityID)) says
    prop (CMD (PR privcmd)) impf prop TRAP)
```

3. The state interpretation function in *configuration* is instantiated with *thermoStateInterp* defined below.
4. The type variable `'output` is instantiated with type *output* defined below.
5. The next-state function *NS* is instantiated with *thermo1NS* defined below.
6. The output function *Out* is instantiated with *thermo1Out* defined below.
7. The certificate list in *configuration* is instantiated with the high-level certificate list *certs*, as defined in Figure 7.7.
8. The authentication function in *configuration* is instantiated with *isAuthenticated* defined below.

In the subsections immediately below, we describe each of the instantiations that have not yet been defined. We then present theorems corresponding to each of the three transition types, *discard, exec*, and *trap*, specialized to the thermostat.

States and Operating Modes

The thermostat has two operating modes: (1) the *Utility* is *enabled* to execute privileged instructions, or (2) the *Utility* is *disabled* to execute privileged instructions. This is defined in HOL by the datatype *mode*.

We define the thermostat's *state* as its operating mode and its temperature setting, which we model as a natural number `num` in HOL. The type definitions of *mode* and *state* are in Figure 7.8.

State Interpretation Function

The interpretation function for thermostat states is given by *thermoStateInterp_def* in Figure 7.9. The definition covers both operating modes:

1. When the operating mode is *enabled*, then the *Utility* has authority to execute privileged commands.

```
thermoStateInterp utilityID privcmd (State enabled temp) =
Name (Role (Utility utilityID)) controls prop (CMD (PR privcmd))
```

2. When the operating mode is *disabled*, then the *Utility's* attempt to execute any privileged instruction is trapped.

```
thermoStateInterp utilityID privcmd (State disabled temp) =
Name (Role (Utility utilityID)) says prop (CMD (PR privcmd)) impf
 prop TRAP
```

The combination of *thermoStateInterp* with the nine access-control logic formulas in *cert*, defined in Figure 7.7, gives the overall security context for the thermostat's SSM to authorize authenticated commands.

Figure 7.10 Transition Diagram for Owner

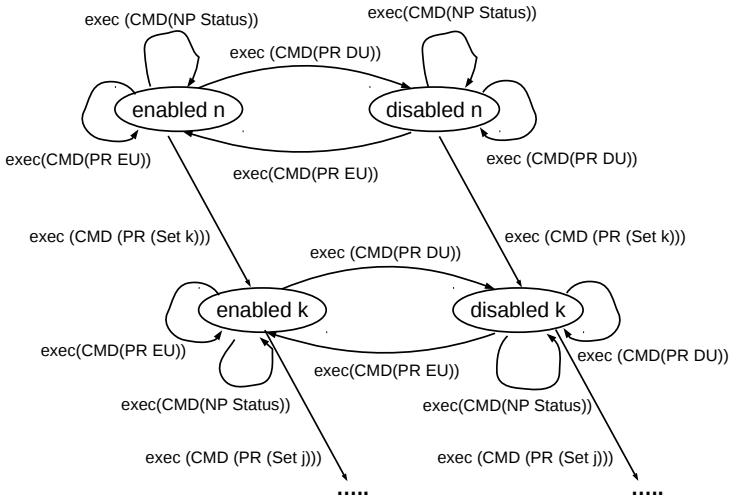
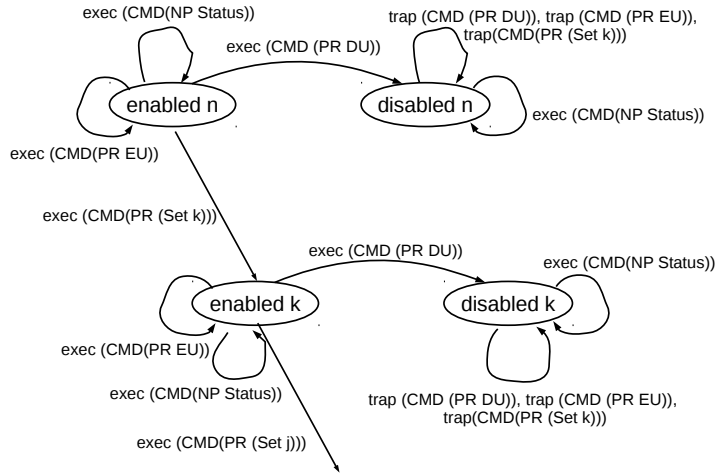


Figure 7.11 Transition Diagram for Utility



Next-State Function

The next-state transition function for the thermostat can be viewed from the standpoints of the *Owner* and *Utility*. Figures 7.10 and 7.11 are the state-transition diagrams for *Owner* and *Utility* originated commands, respectively. Figure 7.10 shows that *Owners* are authorized to execute any command in any state. In particular, they can change the temperature settings and enable or disable the authority of *Utilities* to execute privileged commands.

Figure 7.11 show that *Utilities* can execute privileged commands only if their authority is enabled, i.e., the state mode is *enabled*. If *Utilities* attempt to execute a privileged command in a *disabled* mode, the attempt is trapped. As privileged commands are commands that can change the thermostat's state, i.e., changing either a temperature or mode value, trapped commands result in no state change.

Figure 7.12 shows the definition of *thermo1NS*, the next-state function for the thermostat. If the input is an *exec (CMD cmd)*, then the command results in the appropriate state change or status report. If the input is *trap (CMD cmd)*, then no state change occurs.

Figure 7.13 shows two theorems *npriv_Safe* and *privcmd_Security_Sensitive*. The first theorem states that the next-state function *thermo1NS* has the property that all non-privileged commands *NP npriv* result in no state change. The

Figure 7.12 Next-State Function for Thermostat

```
[thermo1NS_def]

⊢ (thermo1NS (State opMode temp) discard = State opMode temp) ∧
   (thermo1NS (State opMode temp)
      (exec (CMD (PR (Set newTemp)))) =
    State opMode newTemp) ∧
   (thermo1NS (State opMode temp) (exec (CMD (PR EU))) =
    State enabled temp) ∧
   (thermo1NS (State opMode temp) (exec (CMD (PR DU))) =
    State disabled temp) ∧
   (thermo1NS (State opMode temp) (exec (CMD (NP Status))) =
    State opMode temp) ∧
   (thermo1NS (State opMode temp)
      (trap (CMD (PR (Set newTemp)))) =
    State opMode temp) ∧
   (thermo1NS (State opMode temp) (trap (CMD (PR EU))) =
    State opMode temp) ∧
   (thermo1NS (State opMode temp) (trap (CMD (PR DU))) =
    State opMode temp)
```

Figure 7.13 Security Properties of Thermostat Commands

```
[npriv_Safe]

⊢ ∀npriv state. thermo1NS state (exec (CMD (NP npriv))) = state

[privcmd_Security_Sensitive]

⊢ ∀privcmd.
     ∃state. thermo1NS state (exec (CMD (PR privcmd))) ≠ state
```

second theorem states that for all privileged commands *PR privcmd* that a state change is possible when executing the privileged command. These two theorems prove the non-privileged commands are safe, when safety is defined as no change in temperature or operating mode, and that the privileged commands have the capability to change either mode or temperature.

Input Authentication Function

The input authentication function for the top-level thermostat SSM, *isAuthenticated*, is defined by the HOL source code in Figure 7.14. Recall, the top-level SSM uses only access-control logic formulas for inputs and certificates. Given the use cases, there are only three forms of access-control logic formulas that are authenticated:

1. *Keyboard* | *Owner* says $\langle inst \rangle$, i.e., instructions entered in on the attached keyboard,

2. *Server* | *Owner* says $\langle inst \rangle$, i.e., the Server relaying instructions from the Owner, and

3. *Server* | *Utility* says $\langle inst \rangle$, i.e., the Server relaying instructions from the Utility.

All other forms of access-control logic formulas are not authenticated. The three cases above correspond to the first three clauses in the definition in Figure 7.14. The last clause of the definition containing `isAuthenticate _ = F` is interpreted by HOL as all other forms as input produce `F` as an output. The resulting definition is quite long and appears in Appendix E.

Figure 7.14 HOL Source Code Defining isAuthenticated

```
val isAuthenticated_def =
Define
`(isAuthenticated
  ((((Name Keyboard) quoting (Name (Owner ownerID)))  says
  (prop (CMD (cmd:command)))):(command inst, principal,'d,'e)Form) = T) /\
(isAuthenticated
  ((((Name (Key(pubK Server))) quoting (Name (Owner ownerID)))  says
  (prop (CMD (cmd:command)))):(command inst, principal,'d,'e)Form) = T) /\
(isAuthenticated
  ((((Name (Key(pubK Server))) quoting (Name ((Role (Utility utilityID)))))  says
  (prop (CMD (cmd:command)))):(command inst, principal,'d,'e)Form) = T) /\
(isAuthenticated _ = F)`
```

Output Type and Output Function

Figure 7.15 Definition of Thermostat Output type and Output Function

```
output = report state | flag command | null

[thermo1Out_def]

⊢ (thermo1Out (State enabled temp)
      (exec (CMD (PR (Set newTemp)))) =
   report (State enabled newTemp)) ∧
  (thermo1Out (State disabled temp)
      (exec (CMD (PR (Set newTemp)))) =
   report (State disabled newTemp)) ∧
  (thermo1Out (State enabled temp) (exec (CMD (PR EU))) =
   report (State enabled temp)) ∧
  (thermo1Out (State disabled temp) (exec (CMD (PR EU))) =
   report (State enabled temp)) ∧
  (thermo1Out (State enabled temp) (exec (CMD (PR DU))) =
   report (State disabled temp)) ∧
  (thermo1Out (State disabled temp) (exec (CMD (PR DU))) =
   report (State disabled temp)) ∧
  (thermo1Out (State enabled temp) (exec (CMD (NP Status))) =
   report (State enabled temp)) ∧
  (thermo1Out (State disabled temp) (exec (CMD (NP Status))) =
   report (State disabled temp)) ∧
  (thermo1Out (State enabled temp)
      (trap (CMD (PR (Set newTemp)))) =
   flag (PR (Set newTemp))) ∧
  (thermo1Out (State disabled temp)
      (trap (CMD (PR (Set newTemp)))) =
   flag (PR (Set newTemp))) ∧
  (thermo1Out (State enabled temp) (trap (CMD (PR EU))) =
   flag (PR EU)) ∧
  (thermo1Out (State disabled temp) (trap (CMD (PR EU))) =
   flag (PR EU)) ∧
  (thermo1Out (State enabled temp) (trap (CMD (PR DU))) =
   flag (PR DU)) ∧
  (thermo1Out (State disabled temp) (trap (CMD (PR DU))) =
   flag (PR DU)) ∧
  (thermo1Out (State enabled temp) discard = null) ∧
  (thermo1Out (State disabled temp) discard = null)
```

Figure 7.15 shows the definitions of the thermostat output type *output* and the output function *thermo1Out*. There are three kinds of outputs:

1. reporting a state,

2. flagging a command, and

3. null.

Whenever a command is executed, then the new state is reported as output. If a command is trapped, then the command is flagged. If an input is discarded, the *null* is output.

Transition Theorems

Figure 7.16 Configuration Interpretation Justifies Executing Keyboarded Command

[CFGInterpret_Owner_Keyboard_thm]

```
⊢ ∀M Oi Os.
    CFGInterpret (M,Oi,Os)
      (CFG isAuthenticated
         (thermoStateInterp utilityID privcmd)
         (certs ownerID utilityID cmd npriv privcmd)
         (Name Keyboard quoting Name (Owner ownerID) says
          prop (CMD cmd)::ins) s outs) ⇒
    (M,Oi,Os) sat prop (CMD cmd)
```

Figure 7.17 Executing Keyboarded Commands is Justified

[exec_Keyboard_Owner_cmd_Justified]

```
⊢ ∀NS Out outs s ins npriv privcmd cmd ownerID utilityID M Oi
    Os.
    TR (M,Oi,Os) (exec (CMD cmd))
      (CFG isAuthenticated
         (thermoStateInterp utilityID privcmd)
         (certs ownerID utilityID cmd npriv privcmd)
         (Name Keyboard quoting Name (Owner ownerID) says
          prop (CMD cmd)::ins) s outs)
      (CFG isAuthenticated
         (thermoStateInterp utilityID privcmd)
         (certs ownerID utilityID cmd npriv privcmd) ins
         (NS s (exec (CMD cmd)))
         (Out s (exec (CMD cmd))::outs)) ⇒
    (M,Oi,Os) sat prop (CMD cmd)
```

Figure 7.18 Configuration Interpretation Justifies Executing Owner's Command Via Server

[CFGInterpret_Owner_KServer_thm]

```
⊢ ∀M Oi Os.
    CFGInterpret (M,Oi,Os)
      (CFG isAuthenticated
         (thermoStateInterp utilityID privcmd)
         (certs ownerID utilityID cmd npriv privcmd)
         (Name (Key (pubK Server)) quoting
          Name (Owner ownerID) says prop (CMD cmd)::ins) s
         outs) ⇒
    (M,Oi,Os) sat prop (CMD cmd)
```

Figure 7.19 Executing Owner Command Via Server is Justified

[exec_KServer_Owner_cmd_Justified]

```
⊢ ∀NS Out outs s ins npriv privcmd cmd ownerID utilityID M Oi
    Os.
    TR (M,Oi,Os) (exec (CMD cmd))
      (CFG isAuthenticated
         (thermoStateInterp utilityID privcmd)
         (certs ownerID utilityID cmd npriv privcmd)
         (Name (Key (pubK Server)) quoting
          Name (Owner ownerID) says prop (CMD cmd)::ins) s
         outs)
      (CFG isAuthenticated
         (thermoStateInterp utilityID privcmd)
         (certs ownerID utilityID cmd npriv privcmd) ins
         (NS s (exec (CMD cmd)))
         (Out s (exec (CMD cmd))::outs)) ⇒
    (M,Oi,Os) sat prop (CMD cmd)
```

Figure 7.20 Configuration Interpretation Justifies Executing Innocuous Utility Command Via Server

[CFGInterpret_Utility_KServer_npriv_thm]

```
⊢ ∀M Oi Os.
    CFGInterpret (M,Oi,Os)
      (CFG isAuthenticated
         (thermoStateInterp utilityID privcmd)
         (certs ownerID utilityID cmd npriv privcmd)
         (Name (Key (pubK Server)) quoting
          Name (Role (Utility utilityID)) says
          prop (CMD (NP npriv))::ins) s outs) ⇒
    (M,Oi,Os) sat prop (CMD (NP npriv))
```

Figure 7.21 Executing Utility Innocuous Command Via Server is Justified

[exec_KServer_Utility_npriv_Justified]

```
⊢ ∀NS Out outs s ins npriv privcmd cmd ownerID utilityID M Oi
    Os.
  TR (M,Oi,Os) (exec (CMD (NP npriv)))
    (CFG isAuthenticated
       (thermoStateInterp utilityID privcmd)
       (certs ownerID utilityID cmd npriv privcmd)
       (Name (Key (pubK Server)) quoting
        Name (Role (Utility utilityID)) says
        prop (CMD (NP npriv))::ins) s outs)
    (CFG isAuthenticated
       (thermoStateInterp utilityID privcmd)
       (certs ownerID utilityID cmd npriv privcmd) ins
       (NS s (exec (CMD (NP npriv))))
       (Out s (exec (CMD (NP npriv)))::outs)) ⇒
  (M,Oi,Os) sat prop (CMD (NP npriv))
```

Figure 7.22 Configuration Interpretation Justifies Executing Utility Privileged Command

[CFGInterpret_Utility_KServer_privcmd_thm]

```
⊢ ∀M Oi Os.
  CFGInterpret (M,Oi,Os)
    (CFG isAuthenticated
       (thermoStateInterp utilityID privcmd)
       (certs ownerID utilityID cmd npriv privcmd)
       (Name (Key (pubK Server)) quoting
        Name (Role (Utility utilityID)) says
        prop (CMD (PR privcmd))::ins)
       (State enabled temperature) outs) ⇒
  (M,Oi,Os) sat prop (CMD (PR privcmd))
```

Figure 7.23 Executing Utility Privileged Command Via Server is Justified

[exec_KServer_Utility_privcmd_Justified]

```
⊢ ∀NS Out outs temperature ins npriv privcmd cmd ownerID
    utilityID M Oi Os.
  TR (M,Oi,Os) (exec (CMD (PR privcmd)))
    (CFG isAuthenticated
       (thermoStateInterp utilityID privcmd)
       (certs ownerID utilityID cmd npriv privcmd)
       (Name (Key (pubK Server)) quoting
        Name (Role (Utility utilityID)) says
        prop (CMD (PR privcmd))::ins)
       (State enabled temperature) outs)
    (CFG isAuthenticated
       (thermoStateInterp utilityID privcmd)
       (certs ownerID utilityID cmd npriv privcmd) ins
       (NS (State enabled temperature)
          (exec (CMD (PR privcmd))))
       (Out (State enabled temperature)
          (exec (CMD (PR privcmd)))::outs)) ⇒
  (M,Oi,Os) sat prop (CMD (PR privcmd))
```

Figure 7.24 Configuration Interpretation Justifies Trapping Utility Privileged Command

[CFGInterpret_Utility_KServer_trap_thm]

```
⊢ ∀M Oi Os.
  CFGInterpret (M,Oi,Os)
    (CFG isAuthenticated
       (thermoStateInterp utilityID privcmd)
       (certs ownerID utilityID cmd npriv privcmd)
       (Name (Key (pubK Server)) quoting
        Name (Role (Utility utilityID)) says
        prop (CMD (PR privcmd))::ins)
       (State disabled temperature) outs) ⇒
  (M,Oi,Os) sat prop TRAP
```

Figure 7.25 Trapping Utility Privileged Command Via Server is Justified

```
[trap_KServer_Utility_privcmd_Justified]

⊢ ∀NS Out outs temperature ins npriv privcmd cmd ownerID
    utilityID M Oi Os.
  TR (M,Oi,Os) (trap (CMD (PR privcmd)))
    (CFG isAuthenticated
       (thermoStateInterp utilityID privcmd)
       (certs ownerID utilityID cmd npriv privcmd)
       (Name (Key (pubK Server)) quoting
        Name (Role (Utility utilityID)) says
        prop (CMD (PR privcmd))::ins)
       (State disabled temperature) outs)
    (CFG isAuthenticated
       (thermoStateInterp utilityID privcmd)
       (certs ownerID utilityID cmd npriv privcmd) ins
       (NS (State disabled temperature)
          (trap (CMD (PR privcmd))))
       (Out (State disabled temperature)
          (trap (CMD (PR privcmd)))::outs)) ⇒
  (M,Oi,Os) sat prop TRAP
```

We present five configuration theorems and five transition theorems that characterize the behavior of the thermostat. All justify the properties that state if a SSM transition occurred corresponding to executing or trapping an instruction, then it was justified by the security interpretation of the starting configuration.

Figures 7.16, 7.18, 7.20, 7.22, and 7.24 show that executing or trapping an instruction is derivable from the security interpretation provided by *CFGInterpret* applied to the starting configuration. These theorems show that the certificates, state interpretation, and input, do in fact justify executing or trapping an instruction. In other words, that the SSM's actions correspond to sound inference rules.

For example, the theorem *CFGInterpret_Utility_KServer_privcmd_thm* in Figure 7.22 states that the security interpretation of the starting configuration justifies executing the privileged command requested by the *Utility* via the *Server*. Specifically, `(M,Oi,Os) sat prop (CMD (PR privcmd))` is derivable from the interpretation `CFGInterpret` applied to the configuration shown below.

```
[CFGInterpret_Utility_KServer_privcmd_thm]

⊢ ∀M Oi Os.
    CFGInterpret (M,Oi,Os)
      (CFG isAuthenticated
         (thermoStateInterp utilityID privcmd)
         (certs ownerID utilityID cmd npriv privcmd)
         (Name (Key (pubK Server)) quoting
          Name (Role (Utility utilityID)) says
          prop (CMD (PR privcmd))::ins)
         (State enabled temperature) outs) ⇒
    (M,Oi,Os) sat prop (CMD (PR privcmd))
```

The theorems in Figures 7.16, 7.18, 7.20, 7.22, and 7.24 in conjunction with the *TR_exec_cmd_rule* theorem, previously proved in Figure 6.6, give rise to the execution and trap theorems in Figures 7.17, 7.19, 7.21, 7.23, and 7.25.

As an example, the *CFGInterpret_Utility_KServer_privcmd_thm* theorem, *exec_KServer_Utility_privcmd_Justified* shown below and in Figure 7.23. The theorem states that if a *TR* transition occurs corresponding to executing a privileged instruction, i.e., `(exec (CMD (PR privcmd)))`, then executing command was necessarily authorized.

```
[exec_KServer_Utility_privcmd_Justified]
```

```
⊢ ∀NS Out outs temperature ins npriv privcmd cmd ownerID
      utilityID M Oi Os.
    TR (M,Oi,Os) (exec (CMD (PR privcmd)))
      (CFG isAuthenticated
         (thermoStateInterp utilityID privcmd)
         (certs ownerID utilityID cmd npriv privcmd)
         (Name (Key (pubK Server)) quoting
          Name (Role (Utility utilityID)) says
          prop (CMD (PR privcmd))::ins)
         (State enabled temperature) outs)
      (CFG isAuthenticated
         (thermoStateInterp utilityID privcmd)
         (certs ownerID utilityID cmd npriv privcmd) ins
         (NS (State enabled temperature)
            (exec (CMD (PR privcmd))))
         (Out (State enabled temperature)
            (exec (CMD (PR privcmd)))::outs)) ⇒
    (M,Oi,Os) sat prop (CMD (PR privcmd))
```

7.6 Refined Thermostat Secure State-Machine

The refined thermostat SSM is an instantiation of the refined secure state-machine description in Section 6.3. In addition to the instantiations for the top-level thermostat SSM description in Section 7.5, we refine the top-level description by instantiating the following:

1. algebraic types for orders (commands with principals) and messages (orders sent with digital signatures over the network or from the keyboard),

2. an integrity-checking function *checkmsg*,

3. a message interpretaton function *msgInterpret*,

4. algebraic types for certificates used to specify the security context,

5. a message integrity-checking function *checkmsg*, which checks digital signatures, and

6. an interpretation function *cert2Interpret*.

Figure 7.26 Definitions and Theorems for Orders and Messages

```
order = ORD keyPrinc principal command

[order_one_one]

⊢ ∀a0 a1 a2 a'0 a'1 a'2.
    (ORD a0 a1 a2 = ORD a'0 a'1 a'2) ⟺
    (a0 = a'0) ∧ (a1 = a'1) ∧ (a2 = a'2)

msg =
    KB num command
  | MSG keyPrinc principal order
        ((order digest, keyPrinc) asymMsg)

[msg_distinct_thm]

⊢ ∀a3 a2 a'1 a1 a'0 a0. KB a0 a1 ≠ MSG a'0 a'1 a2 a3

[msg_one_one]

⊢ (∀a0 a1 a'0 a'1.
     (KB a0 a1 = KB a'0 a'1) ⟺ (a0 = a'0) ∧ (a1 = a'1)) ∧
   ∀a0 a1 a2 a3 a'0 a'1 a'2 a'3.
     (MSG a0 a1 a2 a3 = MSG a'0 a'1 a'2 a'3) ⟺
     (a0 = a'0) ∧ (a1 = a'1) ∧ (a2 = a'2) ∧ (a3 = a'3)
```

Orders Continuing the development of a secure networked thermostat, we add the definition of an *order* to the definitions and properties of *commands* and *principals*. The purpose of the *order* type is to add authentication and authorization to commands received via the network. This is done by including information on the principals sending commands to thermostats and on whose behalf the senders are acting. Figure 7.26 shows that an *order* has three components:

1. A *keyPrinc* that is sending the message.
2. A *principal* on whose behalf the *keyPrinc* is acting.
3. A *command* issued to the thermostat, e.g., `ORD Server (Role (Utility` *utilityID*`)) (PR (Set` *temperature*`))`—the *Server* passing on a *Set temperature* command from *Utility utilityID*.

The theorem *order_one_one* states that two orders are the same if and only if their components are the same.

Messages We finally define the type *msg* as shown in Figure 7.26. A message received by a thermostat has two sources:

1. the attached *keyboard* from a thermostat associated with an *ownerID* number, e.g., `KB` *userID* `(NP Status)`, and
2. the *Server* sending commands from the *Owner* or *Utility* using the network, e.g.,

```
MSG Server (Role(Utility utilityID))
     (ORD Server (Role(Utility utilityID))(NP Status))
     signature
```

where the `signature` is obtained by signing the hash of the order using the *Server's* private key, i.e.,

```
sign
 (privK Server)
 (hash
  (SOME
   (ORD Server (Role(Utility utilityID)) (NP Status))))
```

The theorems *order_one_one*, *msg_distinct_thm*, and *msg_one_one* are similar to their counterparts for other types. The *distinct* theorem states that network messages are distinct from keyboard messages. The *one_one* theorems state that two orders or messages are the same if and only if their corresponding components are the same.

Authenticating and Checking the Integrity of Messages

Figure 7.27 Authenticating and Checking the Integrity of Messages

```
[checkmsg_def]

⊢ (checkmsg
     (MSG sender recipient (ORD originator role cmd)
        signature) ⟺
   signVerify (pubK sender) signature
     (SOME (ORD originator role cmd)) ∧
   (sender = originator)) ∧ (checkmsg (KB ownerID cmd) ⟺ T)

[checkmsg_OK]

⊢ ((∀ownerID sender recipient originator role cmd.
      (sender = originator) ⇒
      checkmsg
        (MSG sender recipient (ORD originator role cmd)
           (sign (privK sender)
              (hash (SOME (ORD originator role cmd)))))) ∧
   ∀ownerID sender recipient originator role cmd.
     sender ≠ originator ⇒
     ¬checkmsg
        (MSG sender recipient (ORD originator role cmd)
           (sign (privK sender)
              (hash (SOME (ORD originator role cmd)))))) ∧
  ∀ownerID cmd. checkmsg (KB ownerID cmd)
```

Now that the format and contents of messages received by the thermostat are formally defined, we are able to define how messages, orders, and commands are authenticated and checked for integrity. Figure 7.27 shows the definition of *checkmsg* and the theorem *checkmsg_OK*, where *checkmsg_OK* shows that *checkmsg* has the properties we expect.

Looking at the definition of *checkmsg*, we see three things:

1. *checkmsg* applied to orders sent over the network via the *Server* are checked using cryptographic-based digital signatures. Specifically, the digest of the received order is compared against the digest of the original order encrypted using the private key of the sender. This comparison is done using the previously defined cryptographic operation *signVerify*.

2. *checkmsg* as defined requires the *sender* value in the message to match the *originator* value in the order. Of course, there are other definitions of integrity where this might not be the case. We take this approach only as one example out of many.

3. *checkmsg* applied to commands originating from the attached keyboard are assumed to be authentic, i.e., only the owner or people with the owner's permission are able to enter commands in manually. Hence, the value of *checkmsg* applied to keyboard-mediated commands is always true. This is only one possible approach. There

are many possible approaches including biometric-based authentication. For reasons of simplicity and brevity, we assume only *Owners* or their delegates have physical access to a thermostat's keyboard.

The theorem *checkmsg_OK* reflects the design decisions on integrity-checking policy and assumptions contained in *checkmsg*.

1. When the *sender* and *originator* match on messages received over the network from the *Server*, and the digital signature is generated as expected using the previously defined cryptographic operation *sign*, then *checkmsg* will be true, indicating a the received message is intact and authenticated.

2. When the *sender* and *originator* do not match, even when the digital signature is generated as expected, *checkmsg* will be false, indicating the message is not authenticated.

3. Any well-formed keyboard input is regarded as authentic. This reflections the assumption that only the *Owner* or the *Owner's* delegates have physical access to the thermostat's keyboard.

Section 7.6, which follows below, defines the meaning of authenticated messages in the access-control logic. A precise definition of the semantics of messages is essential for assuring a unified view of security among all levels of abstraction.

Interpreting Messages

Figure 7.28 Message Semantics

```
[msgInterpret_def]

⊢ (msgInterpret
      (MSG sender recipient (ORD originator role cmd)
         signature) =
    if
      checkmsg
        (MSG sender recipient (ORD originator role cmd)
           signature)
    then
      Name (Key (pubK sender)) quoting Name role says
      prop (CMD cmd)
    else TT) ∧
   (msgInterpret (KB ownerID cmd) =
    if checkmsg (KB ownerID cmd) then
      Name Keyboard quoting Name (Owner ownerID) says
      prop (CMD cmd)
    else TT)
```

Figure 7.29 Message Interpretation Theorems

[msgInterpretKB]

```
⊢ (M, Oi, Os) sat msgInterpret (KB ownerID cmd) ⟺
   (M, Oi, Os) sat
   Name Keyboard quoting Name (Owner ownerID) says
   prop (CMD cmd)
```

[msgInterpretMSG_sender_originator_match]

```
⊢ msgInterpret
     (MSG sender recipient (ORD sender role cmd)
        (sign (privK sender)
           (hash (SOME (ORD sender role cmd))))) =
   Name (Key (pubK sender)) quoting Name role says
   prop (CMD cmd)
```

[msgInterpretMSG_denied]

```
⊢ sender ≠ originator ⇒
   (msgInterpret
      (MSG sender recipient (ORD originator role cmd)
         (sign (privK sender)
            (hash (SOME (ORD originator role cmd))))) =
    TT)
```

The formal infrastructure of the access-control logic and algebraic models of idealized cryptographic operations accounts for authentication and authorization within the defined interpretation of messages. Figure 7.28 shows the theorem *msgInterpret_def*, which is defines the interpretation or meaning of messages thermostats receive either from the network or from their keyboards. The function *msgInterpret* is defined over the two forms of type *msg*:

1. `MSG sender recipient (ORD originator role cmd) signature)`, i.e., messages from the network, which are expected to be cryptographically signed, and

2. `KB ownerID cmd`, i.e., messages coming directly from a thermostat's keyboard.

In either `MSG` or `KB` messages, first, the incoming message is checked using *checkmsg*, and if the message passes the integrity check, the message's non-trivial meaning in the access-control logic is given. If the message fails *checkmsg*, then the assigned meaning is the trivial assumption `TT` in the access-control logic. Recall `KB` messages that are well-formed are always authenticated. `MSG` messages are authenticated using their digital signatures and verify that the sender and originator are the same.

If an `MSG` message is authenticated, then its interpretation is

`Name (Key (pubK` *sender*`)) quoting Name` *role* `says prop (CMD` *cmd*`)`

If a `KB` message is authenticated, then its interpretation is

`Name Keyboard quoting Name (Owner` *ownerID*`) says prop (CMD` *cmd*`)`

Thermostat Certificates

Figure 7.30 Structure and Integrity-Checking of Thermostat Security Certificates

```
cert2 =
    RCtrCert principal command
  | RRepsCert principal principal command
  | RCtrKCert keyPrinc keyPrinc keyPrinc
  | RKeyCert keyPrinc keyPrinc
  | KeyCert keyPrinc keyPrinc (keyPrinc pKey)
            (((keyPrinc × keyPrinc pKey) digest, keyPrinc)
             asymMsg)

[checkcert2_def]

⊢ (checkcert2 (RCtrCert P cmd) ⟺ T) ∧
   (checkcert2 (RRepsCert P Q cmd) ⟺ T) ∧
   (checkcert2 (RCtrKCert keyPpr Kq keyQpr) ⟺ T) ∧
   (checkcert2 (RKeyCert kp keyPpr) ⟺ T) ∧
   (checkcert2 (KeyCert CApr Ppr (pubK Rpr) signature) ⟺
    signVerify (pubK CApr) signature (SOME (Ppr,pubK Rpr)))
```

Figure 7.31 Interpretation of Thermostat Security Certificates

```
[cert2Interpret_def]

⊢ (cert2Interpret (RCtrCert P cmd) =
    if checkcert2 (RCtrCert P cmd) then
      Name P controls prop (CMD cmd)
    else TT) ∧
   (cert2Interpret (RRepsCert P Q cmd) =
    if checkcert2 (RRepsCert P Q cmd) then
      reps (Name P) (Name Q) (prop (CMD cmd))
    else TT) ∧
   (cert2Interpret (RCtrKCert ca keyKpr keyPpr) =
    if checkcert2 (RCtrKCert ca keyKpr keyPpr) then
      Name (Role ca) controls
      Name (Key (pubK keyKpr)) speaks_for Name (Role keyPpr)
    else TT) ∧
   (cert2Interpret (RKeyCert kppr ca) =
    if checkcert2 (RKeyCert kppr ca) then
      Name (Key (pubK kppr)) speaks_for Name (Role ca)
    else TT) ∧
   (cert2Interpret (KeyCert ca keyPpr (pubK keyRpr) signature) =
    if
      checkcert2 (KeyCert ca keyPpr (pubK keyRpr) signature)
    then
      Name (Key (pubK ca)) says
      Name (Key (pubK keyRpr)) speaks_for Name (Role keyPpr)
    else TT)
```

Figure 7.32 Interpretation Theorems for Security Certificates

```
[cert2InterpretRCtrCert]

⊢ (M,Oi,Os) sat cert2Interpret (RCtrCert (Role P) cmd) ⟺
   (M,Oi,Os) sat Name (Role P) controls prop (CMD cmd)

[cert2InterpretRRepsCert]

⊢ (M,Oi,Os) sat
   cert2Interpret (RRepsCert (Role P) (Role Q) cmd) ⟺
   (M,Oi,Os) sat
   reps (Name (Role P)) (Name (Role Q)) (prop (CMD cmd))

[cert2InterpretRCtrKCert]

⊢ (M,Oi,Os) sat cert2Interpret (RCtrKCert P Q Q) ⟺
   (M,Oi,Os) sat
   Name (Role P) controls
   Name (Key (pubK Q)) speaks_for Name (Role Q)

[cert2InterpretRKeyCert]

⊢ (M,Oi,Os) sat cert2Interpret (RKeyCert P P) ⟺
   (M,Oi,Os) sat Name (Key (pubK P)) speaks_for Name (Role P)

[cert2InterpretKeyCert]

⊢ (M,Oi,Os) sat
   cert2Interpret
     (KeyCert ca P (pubK P)
        (sign (privK ca) (hash (SOME (P,pubK P))))) ⟺
   (M,Oi,Os) sat
   Name (Key (pubK ca)) says
   Name (Key (pubK P)) speaks_for Name (Role P)
```

All commands to the thermostat, which are packaged within `MSG` or `KB` messages of type *msg*, are evaluated within a security context specified by two kinds of statements:

1. *root* certificates, i.e., root trust assumptions corresponding to access-control logic statements, which are *unsigned* because there is no higher authority than *root*, and
2. *digitally signed* certificates, i.e., statements that have meaning in the access-control logic that are signed using the private-key of an authority, presumably recognized by thermostat.

In a way that is exactly analogous to `MSG` messages, digitally signed certificates are authenticated using their digital signatures. Similar to `KB` messages, which do not have associated signatures, `root` certificates are taken at face value. In our thermostat example, we have four root certificates and one signed certificate.

1. *Root Certificates*
   (a) Command authority, `RCtrCert` *P cmd*, interpreted as

   `Name` *P* `controls prop` (`CMD` *cmd*)

   (b) Delegation certificate, `RRepsCert` *P Q cmd*, interpreted as

   `reps` (`Name` *P*) (`Name` *Q*) (`prop` (`CMD` *cmd*))

   (c) Key authority, `RCtrKCert` *ca keyKpr keyPpr*, interpreted as

   `Name` (`Role` *ca*) `controls Name` (`Key` (`pubK` *keyKpr*)) speaks_for `Name` (`Role` *keyPpr*)

(d) Root key certificate, `RKeyCert` *kppr ca*, interpreted as

`Name` (`Key` (`pubK` *kppr*)) speaks_for `Name` (`Role` *ca*)

2. Signed public-key certificate, `KeyCert` *ca keyPpr* (`pubK` *keyRpr*) *signature*, if authenticated is interpreted as

`Name` (`Key` (`pubK` *ca*)) `says` `Name` (`Key` (`pubK` *keyRpr*)) speaks_for `Name` (`Role` *keyPpr*)

Figure 7.30 defines the type *cert2* of thermostat certificates described above. The theorem *checkcert2_def* in Figure 7.30 defines the integrity-checking function for *cert2* certificates. The four root certificates are taken at face value. Signed key certificates are checked using their digital signatures in exactly the same way as `MSG` messages using the previously defined crypto-function *signVerify*.

Certificate Interpretation Function

Figure 7.31 shows the formal definition in HOL of *cert2Interpret_def*, the theorem defining the mapping of *cert2* certificates in the access-control logic formulas. The definition also appears below. Figure 7.32 shows the corresponding meaning of each certificate in terms of Kripke structures satisfying the access-control logic interpretation of each of the five certificate forms.

```
[cert2Interpret_def]

⊢ (cert2Interpret (RCtrCert P cmd) =
    if checkcert2 (RCtrCert P cmd) then
      Name P controls prop (CMD cmd)
    else TT) ∧
   (cert2Interpret (RRepsCert P Q cmd) =
    if checkcert2 (RRepsCert P Q cmd) then
      reps (Name P) (Name Q) (prop (CMD cmd))
    else TT) ∧
   (cert2Interpret (RCtrKCert ca keyKpr keyPpr) =
    if checkcert2 (RCtrKCert ca keyKpr keyPpr) then
      Name (Role ca) controls
      Name (Key (pubK keyKpr)) speaks_for Name (Role keyPpr)
    else TT) ∧
   (cert2Interpret (RKeyCert kppr ca) =
    if checkcert2 (RKeyCert kppr ca) then
      Name (Key (pubK kppr)) speaks_for Name (Role ca)
    else TT) ∧
   (cert2Interpret (KeyCert ca keyPpr (pubK keyRpr) signature) =
    if
      checkcert2 (KeyCert ca keyPpr (pubK keyRpr) signature)
    then
      Name (Key (pubK ca)) says
      Name (Key (pubK keyRpr)) speaks_for Name (Role keyPpr)
    else TT)
```

Note that a certificate's interpretation in the access-control logic is the trivial assumption `TT`, if it fails its integrity check. Root certificates are not digitally signed, as there is no higher level authority to certify them. These are interpreted at face value, with the assumption that root certificates are loaded into the thermostat under controlled and secure circumstances. Certificates with a digital signature, e.g., *KeyCerts*, have their signatures checked using *signVerify*. This is shown below by *checkcert2_def*.

```
[checkcert2_def]

⊢ (checkcert2 (RCtrCert P cmd) ⟺ T) ∧
   (checkcert2 (RRepsCert P Q cmd) ⟺ T) ∧
   (checkcert2 (RCtrKCert keyPpr Kq keyQpr) ⟺ T) ∧
```

```
(checkcert2 (RKeyCert kp keyPpr) ⟺ T) ∧
(checkcert2 (KeyCert CApr Ppr (pubK Rpr) signature) ⟺
 signVerify (pubK CApr) signature (SOME (Ppr,pubK Rpr)))
```

For example, the interpretation of a root key certificate *RKeyCert* is as follows.

```
[cert2InterpretRKeyCert]

⊢ (M,Oi,Os) sat cert2Interpret (RKeyCert P P) ⟺
   (M,Oi,Os) sat Name (Key (pubK P)) speaks_for Name (Role P)
```

For digitally signed *KeyCerts*, theorem *cert2InterpretKeyCert* shows that key certificates signed as expected are interpreted as expected.

```
[cert2InterpretKeyCert]

⊢ (M,Oi,Os) sat
   cert2Interpret
     (KeyCert ca P (pubK P)
        (sign (privK ca) (hash (SOME (P,pubK P))))) ⟺
   (M,Oi,Os) sat
   Name (Key (pubK ca)) says
   Name (Key (pubK P)) speaks_for Name (Role P)
```

Transition Theorems

Figure 7.33 Configuration Interpretation Justifies Executing Keyboarded Command

```
[CFG2Interpret_Owner_Keyboard_thm]

⊢ ∀M Oi Os.
    CFG2Interpret (M,Oi,Os)
      (CFG2 msgInterpret cert2Interpret isAuthenticated
         (certs2 ownerID utilityID cmd npriv privcmd)
         (thermoStateInterp utilityID privcmd)
         (KB ownerID cmd::ins) state outStream) ⇒
    (M,Oi,Os) sat prop (CMD cmd)
```

Figure 7.34 Executing Keyboarded Commands is Justified

```
[exec2_Keyboard_Owner_cmd_Justified]

⊢ ∀NS Out M Oi Os.
    TR2 (M,Oi,Os) (exec (CMD cmd))
      (CFG2 msgInterpret cert2Interpret isAuthenticated
         (certs2 ownerID utilityID cmd npriv privcmd)
         (thermoStateInterp utilityID privcmd)
         (KB ownerID cmd::ins) state outStream)
      (CFG2 msgInterpret cert2Interpret isAuthenticated
         (certs2 ownerID utilityID cmd npriv privcmd)
         (thermoStateInterp utilityID privcmd) ins
         (NS state (exec (CMD cmd)))
         (Out state (exec (CMD cmd))::outStream)) ⇒
    (M,Oi,Os) sat prop (CMD cmd)
```

Figure 7.35 Configuration Interpretation Justifies Executing Owner's Command Via Server

[CFG2Interpret_Owner_KServer_thm]

```
⊢ ∀M Oi Os.
    CFG2Interpret (M,Oi,Os)
      (CFG2 msgInterpret cert2Interpret isAuthenticated
         (certs2 ownerID utilityID cmd npriv privcmd)
         (thermoStateInterp utilityID privcmd)
         (MSG Server (Owner ownerID)
            (ORD Server (Owner ownerID) cmd)
            (sign (privK Server)
               (hash
                  (SOME (ORD Server (Owner ownerID) cmd))))::
           ins) state outStream) ⇒
    (M,Oi,Os) sat prop (CMD cmd)
```

Figure 7.36 Executing Owner Command Via Server is Justified

[exec2_KServer_Owner_cmd_Justified]

```
⊢ ∀NS Out M Oi Os.
    TR2 (M,Oi,Os) (exec (CMD cmd))
      (CFG2 msgInterpret cert2Interpret isAuthenticated
         (certs2 ownerID utilityID cmd npriv privcmd)
         (thermoStateInterp utilityID privcmd)
         (MSG Server (Owner ownerID)
            (ORD Server (Owner ownerID) cmd)
            (sign (privK Server)
               (hash
                  (SOME (ORD Server (Owner ownerID) cmd))))::
           ins) state outStream)
      (CFG2 msgInterpret cert2Interpret isAuthenticated
         (certs2 ownerID utilityID cmd npriv privcmd)
         (thermoStateInterp utilityID privcmd) ins
         (NS state (exec (CMD cmd)))
         (Out state (exec (CMD cmd))::outStream)) ⇒
    (M,Oi,Os) sat prop (CMD cmd)
```

Figure 7.37 Configuration Interpretation Justifies Executing Innocuous Utility Command Via Server

[CFG2Interpret_Utility_KServer_npriv_thm]

```
⊢ ∀M Oi Os.
    CFG2Interpret (M,Oi,Os)
      (CFG2 msgInterpret cert2Interpret isAuthenticated
         (certs2 ownerID utilityID cmd npriv privcmd)
         (thermoStateInterp utilityID privcmd)
         (MSG Server (Role (Utility utilityID))
            (ORD Server (Role (Utility utilityID)) (NP npriv))
            (sign (privK Server)
               (hash
                  (SOME
                     (ORD Server (Role (Utility utilityID))
                        (NP npriv))))))::ins) state outStream) ⇒
    (M,Oi,Os) sat prop (CMD (NP npriv))
```

Figure 7.38 Executing Utility Innocuous Command Via Server is Justified

[exec2_KServer_Utility_npriv_Justified]

```
⊢ ∀NS Out outStream state ins npriv privcmd cmd ownerID
    utilityID M Oi Os.
  TR2 (M,Oi,Os) (exec (CMD (NP npriv)))
    (CFG2 msgInterpret cert2Interpret isAuthenticated
       (certs2 ownerID utilityID cmd npriv privcmd)
       (thermoStateInterp utilityID privcmd)
       (MSG Server (Role (Utility utilityID))
          (ORD Server (Role (Utility utilityID)) (NP npriv))
          (sign (privK Server)
             (hash
                (SOME
                   (ORD Server (Role (Utility utilityID))
                      (NP npriv))))))::ins) state outStream)
    (CFG2 msgInterpret cert2Interpret isAuthenticated
       (certs2 ownerID utilityID cmd npriv privcmd)
       (thermoStateInterp utilityID privcmd) ins
       (NS state (exec (CMD (NP npriv))))
       (Out state (exec (CMD (NP npriv)))::outStream)) ⇒
  (M,Oi,Os) sat prop (CMD (NP npriv))
```

Figure 7.39 Configuration Interpretation Justifies Executing Utility Privileged Command

[CFG2Interpret_Utility_KServer_privcmd_thm]

```
⊢ ∀M Oi Os.
  CFG2Interpret (M,Oi,Os)
    (CFG2 msgInterpret cert2Interpret isAuthenticated
       (certs2 ownerID utilityID cmd npriv privcmd)
       (thermoStateInterp utilityID privcmd)
       (MSG Server (Role (Utility utilityID))
          (ORD Server (Role (Utility utilityID))
             (PR privcmd))
          (sign (privK Server)
             (hash
                (SOME
                   (ORD Server (Role (Utility utilityID))
                      (PR privcmd))))))::ins)
       (State enabled temperature) outStream) ⇒
  (M,Oi,Os) sat prop (CMD (PR privcmd))
```

Figure 7.40 Executing Utility Privileged Command Via Server is Justified

[exec2_KServer_Utility_privcmd_Justified]

```
⊢ ∀NS Out outStream temperature ins npriv privcmd cmd ownerID
    utilityID M Oi Os.
  TR2 (M,Oi,Os) (exec (CMD (PR privcmd)))
    (CFG2 msgInterpret cert2Interpret isAuthenticated
      (certs2 ownerID utilityID cmd npriv privcmd)
      (thermoStateInterp utilityID privcmd)
      (MSG Server (Role (Utility utilityID))
        (ORD Server (Role (Utility utilityID))
          (PR privcmd))
        (sign (privK Server)
          (hash
            (SOME
              (ORD Server (Role (Utility utilityID))
                (PR privcmd))))))::ins)
      (State enabled temperature) outStream)
    (CFG2 msgInterpret cert2Interpret isAuthenticated
      (certs2 ownerID utilityID cmd npriv privcmd)
      (thermoStateInterp utilityID privcmd) ins
      (NS (State enabled temperature)
        (exec (CMD (PR privcmd))))
      (Out (State enabled temperature)
        (exec (CMD (PR privcmd)))::outStream)) ⇒
  (M,Oi,Os) sat prop (CMD (PR privcmd))
```

Figure 7.41 Configuration Interpretation Justifies Trapping Utility Privileged Command

[CFG2Interpret_trap_Utility_KServer_trap_thm]

```
⊢ ∀M Oi Os.
  CFG2Interpret (M,Oi,Os)
    (CFG2 msgInterpret cert2Interpret isAuthenticated
      (certs2 ownerID utilityID cmd npriv privcmd)
      (thermoStateInterp utilityID privcmd)
      (MSG Server (Role (Utility utilityID))
        (ORD Server (Role (Utility utilityID))
          (PR privcmd))
        (sign (privK Server)
          (hash
            (SOME
              (ORD Server (Role (Utility utilityID))
                (PR privcmd))))))::ins)
      (State disabled temperature) outStream) ⇒
  (M,Oi,Os) sat prop TRAP
```

Figure 7.42 Trapping Utility Privileged Command Via Server is Justified

[trap2_KServer_Utility_privcmd_Justified]

```
⊢ ∀NS Out outStream temperature ins npriv privcmd cmd ownerID
    utilityID M Oi Os.
  TR2 (M,Oi,Os) (trap (CMD (PR privcmd)))
    (CFG2 msgInterpret cert2Interpret isAuthenticated
      (certs2 ownerID utilityID cmd npriv privcmd)
      (thermoStateInterp utilityID privcmd)
      (MSG Server (Role (Utility utilityID))
        (ORD Server (Role (Utility utilityID))
          (PR privcmd))
        (sign (privK Server)
          (hash
            (SOME
              (ORD Server (Role (Utility utilityID))
                (PR privcmd))))))::ins)
      (State disabled temperature) outStream)
    (CFG2 msgInterpret cert2Interpret isAuthenticated
      (certs2 ownerID utilityID cmd npriv privcmd)
      (thermoStateInterp utilityID privcmd) ins
      (NS (State disabled temperature)
        (trap (CMD (PR privcmd))))
      (Out (State disabled temperature)
        (trap (CMD (PR privcmd)))::outStream)) ⇒
  (M,Oi,Os) sat prop TRAP
```

The refined thermostat SSM description has ten theorems characterizing its behavior corresponding to the ten theorems for the top-level SSM. Five of the theorems show that the security interpretation of *configuration2* justifies executing or trapping the particular instructions shown. These theorems are derived inference rules in C2 calculus. The five *configuration2* theorems are Figures 7.33, 7.35, 7.37, 7.39, and 7.41. For example, the theorem *CFG2Interpret_Utility_KServer_privcmd_thm* shows that executing a privileged command at the request of the *Utility* is derivable from the configuration shown in the theorem below. This corresponds exactly to the top-level SSM description, except that the access-control logic formulas corresponding to inputs and certificates is now replaced by input and certificate data structures, and their interpretations.

[CFG2Interpret_Utility_KServer_privcmd_thm]

```
⊢ ∀M Oi Os.
    CFG2Interpret (M,Oi,Os)
      (CFG2 msgInterpret cert2Interpret isAuthenticated
        (certs2 ownerID utilityID cmd npriv privcmd)
        (thermoStateInterp utilityID privcmd)
        (MSG Server (Role (Utility utilityID))
          (ORD Server (Role (Utility utilityID))
            (PR privcmd))
          (sign (privK Server)
            (hash
              (SOME
                (ORD Server (Role (Utility utilityID))
                  (PR privcmd))))))::ins)
        (State enabled temperature) outStream) ⇒
    (M,Oi,Os) sat prop (CMD (PR privcmd))
```

The refined execution theorems corresponding to the top-level SSM execution theorems are in Figures 7.34, 7.36, 7.38, 7.40, and 7.42. As an example, the *exec2_KServer_Utility_privcmd_Justified* is shown below. Similar to its

counterpart in the top-level SSM description, the theorem states that if a transition occurred corresponding to executing a privileged command from the *Utility*, then the execution was justified.

[exec2_KServer_Utility_privcmd_Justified]

```
⊢ ∀NS Out outStream temperature ins npriv privcmd cmd ownerID
      utilityID M Oi Os.
    TR2 (M,Oi,Os) (exec (CMD (PR privcmd)))
      (CFG2 msgInterpret cert2Interpret isAuthenticated
         (certs2 ownerID utilityID cmd npriv privcmd)
         (thermoStateInterp utilityID privcmd)
         (MSG Server (Role (Utility utilityID))
            (ORD Server (Role (Utility utilityID))
               (PR privcmd))
            (sign (privK Server)
               (hash
                  (SOME
                     (ORD Server (Role (Utility utilityID))
                        (PR privcmd)))))::ins)
         (State enabled temperature) outStream)
      (CFG2 msgInterpret cert2Interpret isAuthenticated
         (certs2 ownerID utilityID cmd npriv privcmd)
         (thermoStateInterp utilityID privcmd) ins
         (NS (State enabled temperature)
            (exec (CMD (PR privcmd))))
         (Out (State enabled temperature)
            (exec (CMD (PR privcmd)))::outStream)) ⇒
    (M,Oi,Os) sat prop (CMD (PR privcmd))
```

7.7 Equivalence of Top-Level and Refined Secure State-Machines

Figure 7.43 TR and TR2 Equivalence for Keyboard Commands

[TR2_iff_TR_Keyboard_Owner_cmd]

```
⊢ ∀M Oi Os ownerID utilityID ins ins2 outStream NS Out state
    npriv privcmd cmd.
  TR2 (M,Oi,Os) (exec (CMD cmd))
    (CFG2 msgInterpret cert2Interpret isAuthenticated
       (certs2 ownerID utilityID cmd npriv privcmd)
       (thermoStateInterp utilityID privcmd)
       (KB ownerID cmd::ins2) state outStream)
    (CFG2 msgInterpret cert2Interpret isAuthenticated
       (certs2 ownerID utilityID cmd npriv privcmd)
       (thermoStateInterp utilityID privcmd) ins2
       (NS state (exec (CMD cmd)))
       (Out state (exec (CMD cmd))::outStream)) ⟺
  TR (M,Oi,Os) (exec (CMD cmd))
    (CFG isAuthenticated
       (thermoStateInterp utilityID privcmd)
       (certs ownerID utilityID cmd npriv privcmd)
       (Name Keyboard quoting Name (Owner ownerID) says
        prop (CMD cmd)::ins) state outStream)
    (CFG isAuthenticated
       (thermoStateInterp utilityID privcmd)
       (certs ownerID utilityID cmd npriv privcmd) ins
       (NS state (exec (CMD cmd)))
       (Out state (exec (CMD cmd))::outStream))
```

Figure 7.44 TR and TR2 Equivalence for Owner Commands Via Server

[TR2_iff_TR_KServer_Owner_cmd]

```
⊢ ∀M Oi Os ownerID utilityID ins ins2 outStream NS Out state
    npriv privcmd cmd.
  TR2 (M,Oi,Os) (exec (CMD cmd))
    (CFG2 msgInterpret cert2Interpret isAuthenticated
      (certs2 ownerID utilityID cmd npriv privcmd)
      (thermoStateInterp utilityID privcmd)
      (MSG Server (Owner ownerID)
        (ORD Server (Owner ownerID) cmd)
        (sign (privK Server)
          (hash
            (SOME (ORD Server (Owner ownerID) cmd))))::
        ins2) state outStream)
    (CFG2 msgInterpret cert2Interpret isAuthenticated
      (certs2 ownerID utilityID cmd npriv privcmd)
      (thermoStateInterp utilityID privcmd) ins2
      (NS state (exec (CMD cmd)))
      (Out state (exec (CMD cmd))::outStream)) ⟺
  TR (M,Oi,Os) (exec (CMD cmd))
    (CFG isAuthenticated
      (thermoStateInterp utilityID privcmd)
      (certs ownerID utilityID cmd npriv privcmd)
      (Name (Key (pubK Server)) quoting
       Name (Owner ownerID) says prop (CMD cmd)::ins) state
      outStream)
    (CFG isAuthenticated
      (thermoStateInterp utilityID privcmd)
      (certs ownerID utilityID cmd npriv privcmd) ins
      (NS state (exec (CMD cmd)))
      (Out state (exec (CMD cmd))::outStream))
```

Figure 7.45 TR and TR2 Equivalence for Utility Non-Privileged Commands Via Server

[TR2_iff_TR_KServer_Utility_npriv]

```
⊢ ∀M Oi Os ownerID utilityID ins ins2 outStream NS Out state
    npriv privcmd cmd.
  TR2 (M,Oi,Os) (exec (CMD (NP npriv)))
    (CFG2 msgInterpret cert2Interpret isAuthenticated
      (certs2 ownerID utilityID cmd npriv privcmd)
      (thermoStateInterp utilityID privcmd)
      (MSG Server (Role (Utility utilityID))
        (ORD Server (Role (Utility utilityID)) (NP npriv))
        (sign (privK Server)
          (hash
            (SOME
              (ORD Server (Role (Utility utilityID))
                (NP npriv)))))::ins2) state outStream)
    (CFG2 msgInterpret cert2Interpret isAuthenticated
      (certs2 ownerID utilityID cmd npriv privcmd)
      (thermoStateInterp utilityID privcmd) ins2
      (NS state (exec (CMD (NP npriv))))
      (Out state (exec (CMD (NP npriv)))::outStream)) ⟺
  TR (M,Oi,Os) (exec (CMD (NP npriv)))
    (CFG isAuthenticated
      (thermoStateInterp utilityID privcmd)
      (certs ownerID utilityID cmd npriv privcmd)
      (Name (Key (pubK Server)) quoting
       Name (Role (Utility utilityID)) says
       prop (CMD (NP npriv))::ins) state outStream)
    (CFG isAuthenticated
      (thermoStateInterp utilityID privcmd)
      (certs ownerID utilityID cmd npriv privcmd) ins
      (NS state (exec (CMD (NP npriv))))
      (Out state (exec (CMD (NP npriv)))::outStream))
```

Figure 7.46 TR and TR2 Equivalence for Utility Privileged Commands Via Server

[TR2_iff_TR_KServer_Utility_privcmd]

```
⊢ ∀M Oi Os ownerID utilityID ins ins2 temperature outStream NS
      Out npriv privcmd cmd.
    TR2 (M,Oi,Os) (exec (CMD (PR privcmd)))
      (CFG2 msgInterpret cert2Interpret isAuthenticated
         (certs2 ownerID utilityID cmd npriv privcmd)
         (thermoStateInterp utilityID privcmd)
         (MSG Server (Role (Utility utilityID))
            (ORD Server (Role (Utility utilityID))
               (PR privcmd))
            (sign (privK Server)
               (hash
                  (SOME
                     (ORD Server (Role (Utility utilityID))
                        (PR privcmd))))))::ins2)
         (State enabled temperature) outStream)
      (CFG2 msgInterpret cert2Interpret isAuthenticated
         (certs2 ownerID utilityID cmd npriv privcmd)
         (thermoStateInterp utilityID privcmd) ins2
         (NS (State enabled temperature)
            (exec (CMD (PR privcmd))))
         (Out (State enabled temperature)
            (exec (CMD (PR privcmd)))::outStream)) ⟺
    TR (M,Oi,Os) (exec (CMD (PR privcmd)))
      (CFG isAuthenticated
         (thermoStateInterp utilityID privcmd)
         (certs ownerID utilityID cmd npriv privcmd)
         (Name (Key (pubK Server)) quoting
          Name (Role (Utility utilityID)) says
          prop (CMD (PR privcmd))::ins)
         (State enabled temperature) outStream)
      (CFG isAuthenticated
         (thermoStateInterp utilityID privcmd)
         (certs ownerID utilityID cmd npriv privcmd) ins
         (NS (State enabled temperature)
            (exec (CMD (PR privcmd))))
         (Out (State enabled temperature)
            (exec (CMD (PR privcmd)))::outStream))
```

Figure 7.47 TR and TR2 Equivalence for Trapping Utility Privileged Commands Via Server

[TR2_iff_TR_KServer_Utility_trap]

```
⊢ ∀M Oi Os ownerID utilityID ins ins2 temperature outStream NS
    Out npriv privcmd cmd.
  TR2 (M,Oi,Os) (trap (CMD (PR privcmd)))
    (CFG2 msgInterpret cert2Interpret isAuthenticated
      (certs2 ownerID utilityID cmd npriv privcmd)
      (thermoStateInterp utilityID privcmd)
      (MSG Server (Role (Utility utilityID))
        (ORD Server (Role (Utility utilityID))
          (PR privcmd))
        (sign (privK Server)
          (hash
            (SOME
              (ORD Server (Role (Utility utilityID))
                (PR privcmd))))))::ins2)
      (State disabled temperature) outStream)
    (CFG2 msgInterpret cert2Interpret isAuthenticated
      (certs2 ownerID utilityID cmd npriv privcmd)
      (thermoStateInterp utilityID privcmd) ins2
      (NS (State disabled temperature)
        (trap (CMD (PR privcmd))))
      (Out (State disabled temperature)
        (trap (CMD (PR privcmd)))::outStream)) ⟺
  TR (M,Oi,Os) (trap (CMD (PR privcmd)))
    (CFG isAuthenticated
      (thermoStateInterp utilityID privcmd)
      (certs ownerID utilityID cmd npriv privcmd)
      (Name (Key (pubK Server)) quoting
       Name (Role (Utility utilityID)) says
       prop (CMD (PR privcmd))::ins)
      (State disabled temperature) outStream)
    (CFG isAuthenticated
      (thermoStateInterp utilityID privcmd)
      (certs ownerID utilityID cmd npriv privcmd) ins
      (NS (State disabled temperature)
        (trap (CMD (PR privcmd))))
      (Out (State disabled temperature)
        (trap (CMD (PR privcmd)))::outStream))
```

The last group of theorems for the networked thermostat are five equivalence theorems. For the cases of (1) executing keyboarded commands by the *Owner*, (2) executing *Owner* command via the *Server*, (3) executing non-privileged *Utility* commands via the *Server*, (4) executing privileged *Utility* commands via the *Server*, and (5) trapping privileged commands via the *Server*, the theorems state that the top-level and refined SSM transitions are equivalent. Figures 7.43, 7.44, 7.45, 7.46, and 7.47 have the five theorems.

Chapter 8

Conclusions

The objectives of certified security by design (CSBD) are to

1. give formally verified assurances that all commands are executed if and only if they authenticated and authorized,
2. assure a consistent and unified view of security across all levels of abstraction from high-level CONOPS down to implementations, and
3. enable third parties to rapidly and easily reproduce all formally verified assurance results.

In this chapter we have provided a detailed outline and description of how to do this within a reusable and parameterized design and verification infrastructure consisting of:

1. an access-control logic and command-and-control (C2) calculus based on a multi-agent propositional modal logic with Kripke semantics,
2. an algebraic model of cryptographic operations,
3. secure state-machine models integrating authentication, authorization, security interpretation, next-state, and output functions as parameters in transition relations, and
4. implementations of all of the above as formally verified machine-checked theories in the HOL-4 (Higher Order Logic) theorem prover.

As an illustration, we developed a networked thermostat that incorporated security into all design levels from high-level models down to secure state-machines using specialized message and certificate structures. What is notable is that most of the formal infrastructure is parameterized and reusable. As high-order logic is at the foundation of our methods, we are able to achieve generality by parameterizing over functions such as next-state, output, authentication, authorization, and interpretation functions. All of this leads to the conclusion that formal assurance of command-and-control functions in the Internet of Things is feasible.

Appendix A

HOL Definition of ACL Syntax and Kripke Structures

```
Form =
    TT
  | FF
  | prop 'aavar
  | notf (('aavar, 'apn, 'il, 'sl) Form)
  | (andf) (('aavar, 'apn, 'il, 'sl) Form)
           (('aavar, 'apn, 'il, 'sl) Form)
  | (orf) (('aavar, 'apn, 'il, 'sl) Form)
          (('aavar, 'apn, 'il, 'sl) Form)
  | (impf) (('aavar, 'apn, 'il, 'sl) Form)
           (('aavar, 'apn, 'il, 'sl) Form)
  | (eqf) (('aavar, 'apn, 'il, 'sl) Form)
          (('aavar, 'apn, 'il, 'sl) Form)
  | (says) ('apn Princ) (('aavar, 'apn, 'il, 'sl) Form)
  | (speaks_for) ('apn Princ) ('apn Princ)
  | (controls) ('apn Princ) (('aavar, 'apn, 'il, 'sl) Form)
  | reps ('apn Princ) ('apn Princ)
         (('aavar, 'apn, 'il, 'sl) Form)
  | (domi) (('apn, 'il) IntLevel) (('apn, 'il) IntLevel)
  | (eqi) (('apn, 'il) IntLevel) (('apn, 'il) IntLevel)
  | (doms) (('apn, 'sl) SecLevel) (('apn, 'sl) SecLevel)
  | (eqs) (('apn, 'sl) SecLevel) (('apn, 'sl) SecLevel)
  | (eqn) num num
  | (lte) num num
  | (lt) num num

Kripke =
    KS ('aavar -> 'aaworld -> bool)
       ('apn -> 'aaworld -> 'aaworld -> bool) ('apn -> 'il)
       ('apn -> 'sl)

Princ =
    Name 'apn
  | (meet) ('apn Princ) ('apn Princ)
  | (quoting) ('apn Princ) ('apn Princ) ;

IntLevel = iLab 'il | il 'apn ;

SecLevel = sLab 'sl | sl 'apn
```

Appendix B

HOL Definition of ACL Semantics

The semantics or values of well-formed access-control logic formulas in HOL, is defined by `Efn`. The the values of well-formed access-control logic formulas are sets of worlds that are members of the universe of worlds for a given Kripke structure M.

[Efn_def]

```
⊢ (∀Oi Os M. Efn Oi Os M TT = 𝒰(:'v)) ∧
  (∀Oi Os M. Efn Oi Os M FF = {}) ∧
  (∀Oi Os M p. Efn Oi Os M (prop p) = intpKS M p) ∧
  (∀Oi Os M f.
     Efn Oi Os M (notf f) = 𝒰(:'v) DIFF Efn Oi Os M f) ∧
  (∀Oi Os M f1 f2.
     Efn Oi Os M (f1 andf f2) =
     Efn Oi Os M f1 ∩ Efn Oi Os M f2) ∧
  (∀Oi Os M f1 f2.
     Efn Oi Os M (f1 orf f2) =
     Efn Oi Os M f1 ∪ Efn Oi Os M f2) ∧
  (∀Oi Os M f1 f2.
     Efn Oi Os M (f1 impf f2) =
     𝒰(:'v) DIFF Efn Oi Os M f1 ∪ Efn Oi Os M f2) ∧
  (∀Oi Os M f1 f2.
     Efn Oi Os M (f1 eqf f2) =
     (𝒰(:'v) DIFF Efn Oi Os M f1 ∪ Efn Oi Os M f2) ∩
     (𝒰(:'v) DIFF Efn Oi Os M f2 ∪ Efn Oi Os M f1)) ∧
  (∀Oi Os M P f.
     Efn Oi Os M (P says f) =
     {w | Jext (jKS M) P w ⊆ Efn Oi Os M f}) ∧
  (∀Oi Os M P Q.
     Efn Oi Os M (P speaks_for Q) =
     if Jext (jKS M) Q RSUBSET Jext (jKS M) P then 𝒰(:'v)
     else {}) ∧
  (∀Oi Os M P f.
     Efn Oi Os M (P controls f) =
     𝒰(:'v) DIFF {w | Jext (jKS M) P w ⊆ Efn Oi Os M f} ∪
     Efn Oi Os M f) ∧
  (∀Oi Os M P Q f.
     Efn Oi Os M (reps P Q f) =
     𝒰(:'v) DIFF
     {w | Jext (jKS M) (P quoting Q) w ⊆ Efn Oi Os M f} ∪
     {w | Jext (jKS M) Q w ⊆ Efn Oi Os M f}) ∧
  (∀Oi Os M intl1 intl2.
     Efn Oi Os M (intl1 domi intl2) =
     if repPO Oi (Lifn M intl2) (Lifn M intl1) then 𝒰(:'v)
     else {}) ∧
  (∀Oi Os M intl2 intl1.
     Efn Oi Os M (intl2 eqi intl1) =
     (if repPO Oi (Lifn M intl2) (Lifn M intl1) then 𝒰(:'v)
      else {}) ∩
     if repPO Oi (Lifn M intl1) (Lifn M intl2) then 𝒰(:'v)
```

```
   else { }) ∧
(∀Oi Os M secl1 secl2 .
   Efn Oi Os M (secl1 doms secl2) =
   if repPO Os (Lsfn M secl2) (Lsfn M secl1) then 𝒰(:'v)
   else { }) ∧
(∀Oi Os M secl2 secl1 .
   Efn Oi Os M (secl2 eqs secl1) =
   (if repPO Os (Lsfn M secl2) (Lsfn M secl1) then 𝒰(:'v)
    else { }) ∩
   if repPO Os (Lsfn M secl1) (Lsfn M secl2) then 𝒰(:'v)
   else { }) ∧
(∀Oi Os M numExp1 numExp2 .
   Efn Oi Os M (numExp1 eqn numExp2) =
   if numExp1 = numExp2 then 𝒰(:'v) else { }) ∧
(∀Oi Os M numExp1 numExp2 .
   Efn Oi Os M (numExp1 lte numExp2) =
   if numExp1 ≤ numExp2 then 𝒰(:'v) else { }) ∧
∀Oi Os M numExp1 numExp2 .
  Efn Oi Os M (numExp1 lt numExp2) =
  if numExp1 < numExp2 then 𝒰(:'v) else { }
```

Appendix C

HOL Definition and Properties of Transition Relation TR

C.1 HOL Source Code Defining TR

```
val (TR_rules, TR_ind, TR_cases) =
Hol_reln
`(!(inputTest:('command inst,'principal,'d,'e)Form -> bool) (P:'principal Princ)
    (NS: 'state -> 'command inst trType -> 'state) M Oi Os Out (s:'state)
    (certs:('command inst,'principal,'d,'e)Form list)
    (stateInterp:'state -> ('command inst,'principal,'d,'e)Form)
   (cmd:'command)(ins:('command inst,'principal,'d,'e)Form list)
   (outs:'output list).
(inputTest ((P says (prop (CMD cmd))):('command inst,'principal,'d,'e)Form) /\
 (CFGInterpret (M,Oi,Os)
  (CFG inputTest stateInterp certs (((P says (prop (CMD cmd)))
  :('command inst,'principal,'d,'e)Form)::ins) s outs))) ==>
(TR
 ((M:('command inst,'b,'principal,'d,'e)Kripke),Oi:'d po,Os:'e po) (exec(CMD cmd))
 (CFG inputTest stateInterp certs (((P says (prop (CMD cmd)))
 :('command inst,'principal,'d,'e)Form)::ins) s outs)
 (CFG inputTest stateInterp certs ins (NS s (exec(CMD cmd))) ((Out s (exec(CMD cmd)))::outs)))) /\
(!(inputTest:('command inst,'principal,'d,'e)Form -> bool) (P:'principal Princ)
    (NS:'state -> 'command inst trType -> 'state) M Oi Os Out (s:'state)
    (certs:('command inst,'principal,'d,'e)Form list)
    (stateInterp:'state -> ('command inst,'principal,'d,'e)Form)
   (cmd:'command)(ins:('command inst,'principal,'d,'e)Form list)
   (outs:'output list).
(inputTest ((P says (prop (CMD cmd))):('command inst,'principal,'d,'e)Form) /\
 (CFGInterpret (M,Oi,Os)
  (CFG inputTest stateInterp certs (((P says (prop (CMD cmd)))
  :('command inst,'principal,'d,'e)Form)::ins) s outs))) ==>
(TR
 ((M:('command inst,'b,'principal,'d,'e)Kripke),Oi:'d po,Os:'e po) (trap(CMD cmd))
 (CFG inputTest stateInterp certs (((P says (prop (CMD cmd)))
 :('command inst,'principal,'d,'e)Form)::ins) s outs)
 (CFG inputTest stateInterp certs ins (NS s (trap(CMD cmd))) ((Out s (trap(CMD cmd)))::outs)))) /\
(!(inputTest:('command inst,'principal,'d,'e)Form -> bool) (NS:'state -> 'command inst trType -> 'state)
   M Oi Os (Out: 'state -> 'command inst trType -> 'output) (s:'state)
   (certs:('command inst,'principal,'d,'e)Form list)
   (stateInterp:'state -> ('command inst,'principal,'d,'e)Form)
   (cmd:'command)(x:('command inst,'principal,'d,'e)Form)(ins:('command inst,'principal,'d,'e)Form list)
   (outs:'output list).
~inputTest x ==>
(TR
 ((M:('command inst,'b,'principal,'d,'e)Kripke),Oi:'d po,Os:'e po) (discard:'command inst trType)
 (CFG inputTest stateInterp certs ((x:('command inst,'principal,'d,'e)Form)::ins) s outs)
 (CFG inputTest stateInterp certs ins (NS s discard) ((Out s discard)::outs))))`
```

C.2 Defining Properties of TR

[TR_rules]

$$
\begin{aligned}
\vdash\ & (\forall \mathit{inputTest}\ \ P\ \ NS\ \ M\ \ Oi\ \ Os\ \ Out\ \ s\ \ certs\ \ stateInterp\ \ cmd\ \ ins \\
& \quad \mathit{outs}. \\
& \quad \mathit{inputTest}\ \ (P\ \ \texttt{says prop (CMD}\ \ \mathit{cmd}\texttt{)})\ \ \wedge \\
& \quad \texttt{CFGInterpret}\ \ (M, Oi, Os) \\
& \qquad \texttt{(CFG}\ \ \mathit{inputTest}\ \ \mathit{stateInterp}\ \ \mathit{certs} \\
& \qquad\quad (P\ \ \texttt{says prop (CMD}\ \ \mathit{cmd}\texttt{)::}\mathit{ins})\ \ s\ \ \mathit{outs})\ \ \Rightarrow \\
& \quad \texttt{TR}\ \ (M, Oi, Os)\ \ \texttt{(exec (CMD}\ \ \mathit{cmd}\texttt{))} \\
& \qquad \texttt{(CFG}\ \ \mathit{inputTest}\ \ \mathit{stateInterp}\ \ \mathit{certs} \\
& \qquad\quad (P\ \ \texttt{says prop (CMD}\ \ \mathit{cmd}\texttt{)::}\mathit{ins})\ \ s\ \ \mathit{outs}) \\
& \qquad \texttt{(CFG}\ \ \mathit{inputTest}\ \ \mathit{stateInterp}\ \ \mathit{certs}\ \ \mathit{ins} \\
& \qquad\quad (NS\ \ s\ \ \texttt{(exec (CMD}\ \ \mathit{cmd}\texttt{))}) \\
& \qquad\quad (Out\ \ s\ \ \texttt{(exec (CMD}\ \ \mathit{cmd}\texttt{))::}\mathit{outs}))\ \ \wedge \\
& (\forall \mathit{inputTest}\ \ P\ \ NS\ \ M\ \ Oi\ \ Os\ \ Out\ \ s\ \ certs\ \ stateInterp\ \ cmd\ \ ins
\end{aligned}
$$

```
        outs.
       inputTest (P says prop (CMD cmd)) ∧
       CFGInterpret (M, Oi, Os)
         (CFG inputTest stateInterp certs
            (P says prop (CMD cmd)::ins) s outs) ⇒
       TR (M, Oi, Os) (trap (CMD cmd))
         (CFG inputTest stateInterp certs
            (P says prop (CMD cmd)::ins) s outs)
         (CFG inputTest stateInterp certs ins
            (NS s (trap (CMD cmd)))
            (Out s (trap (CMD cmd))::outs))) ∧
   ∀inputTest NS M Oi Os Out s certs stateInterp cmd x ins outs.
     ¬inputTest x ⇒
     TR (M, Oi, Os) discard
       (CFG inputTest stateInterp certs (x::ins) s outs)
       (CFG inputTest stateInterp certs ins (NS s discard)
          (Out s discard::outs))
```

[TR_ind]

```
⊢ ∀TR'.
    (∀inputTest P NS M Oi Os Out s certs stateInterp cmd ins
       outs.
      inputTest (P says prop (CMD cmd)) ∧
      CFGInterpret (M, Oi, Os)
        (CFG inputTest stateInterp certs
           (P says prop (CMD cmd)::ins) s outs) ⇒
      TR' (M, Oi, Os) (exec (CMD cmd))
        (CFG inputTest stateInterp certs
           (P says prop (CMD cmd)::ins) s outs)
        (CFG inputTest stateInterp certs ins
           (NS s (exec (CMD cmd)))
           (Out s (exec (CMD cmd))::outs))) ∧
    (∀inputTest P NS M Oi Os Out s certs stateInterp cmd ins
       outs.
      inputTest (P says prop (CMD cmd)) ∧
      CFGInterpret (M, Oi, Os)
        (CFG inputTest stateInterp certs
           (P says prop (CMD cmd)::ins) s outs) ⇒
      TR' (M, Oi, Os) (trap (CMD cmd))
        (CFG inputTest stateInterp certs
           (P says prop (CMD cmd)::ins) s outs)
        (CFG inputTest stateInterp certs ins
           (NS s (trap (CMD cmd)))
           (Out s (trap (CMD cmd))::outs))) ∧
    (∀inputTest NS M Oi Os Out s certs stateInterp cmd x ins
       outs.
      ¬inputTest x ⇒
      TR' (M, Oi, Os) discard
        (CFG inputTest stateInterp certs (x::ins) s outs)
        (CFG inputTest stateInterp certs ins (NS s discard)
           (Out s discard::outs))) ⇒
    ∀a0 a1 a2 a3. TR a0 a1 a2 a3 ⇒ TR' a0 a1 a2 a3
```

[TR_cases]

```
⊢ ∀a0 a1 a2 a3.
    TR a0 a1 a2 a3 ⟺
    (∃inputTest P NS M Oi Os Out s certs stateInterp cmd ins
```

```
    outs.
   (a0 = (M, Oi, Os)) ∧ (a1 = exec (CMD cmd)) ∧
   (a2 =
    CFG inputTest stateInterp certs
      (P says prop (CMD cmd)::ins) s outs) ∧
   (a3 =
    CFG inputTest stateInterp certs ins
      (NS s (exec (CMD cmd)))
      (Out s (exec (CMD cmd))::outs)) ∧
  inputTest (P says prop (CMD cmd)) ∧
  CFGInterpret (M, Oi, Os)
    (CFG inputTest stateInterp certs
       (P says prop (CMD cmd)::ins) s outs)) ∨
(∃inputTest P NS M Oi Os Out s certs stateInterp cmd ins
    outs.
   (a0 = (M, Oi, Os)) ∧ (a1 = trap (CMD cmd)) ∧
   (a2 =
    CFG inputTest stateInterp certs
      (P says prop (CMD cmd)::ins) s outs) ∧
   (a3 =
    CFG inputTest stateInterp certs ins
      (NS s (trap (CMD cmd)))
      (Out s (trap (CMD cmd))::outs)) ∧
  inputTest (P says prop (CMD cmd)) ∧
  CFGInterpret (M, Oi, Os)
    (CFG inputTest stateInterp certs
       (P says prop (CMD cmd)::ins) s outs)) ∨
∃inputTest NS M Oi Os Out s certs stateInterp cmd x ins
   outs.
  (a0 = (M, Oi, Os)) ∧ (a1 = discard) ∧
  (a2 = CFG inputTest stateInterp certs (x::ins) s outs) ∧
  (a3 =
   CFG inputTest stateInterp certs ins (NS s discard)
     (Out s discard::outs)) ∧ ¬inputTest x
```

Appendix D

HOL Definition and Properties of Transition Relation TR2

D.1 HOL Source Code Defining TR2

```
val (TR2_rules, TR2_ind, TR2_cases) =
Hol_reln
`(!(inputInterpret: 'input -> ('command inst,'principal,'d,'e)Form)
 (certInterpret: 'cert -> ('command inst,'principal,'d,'e)Form)
 (inputTest:('command inst,'principal,'d,'e)Form -> bool)
 (x:'input)
 (NS: 'state -> 'command inst trType -> 'state)
 (M:('command inst,'b,'principal,'d,'e)Kripke)
 (Oi:'d po)
 (Os:'e po)
 (Out: 'state -> 'command inst trType -> 'output)
 (state:'state)
 (certs:'cert list)
 (stateInterpret:'state -> ('command inst,'principal,'d,'e)Form)
 (cmd:'command)
 (ins:'input list)
 (outStream:'output list).
(inputTest(inputInterpret (x:'input))) /\
(CFG2Interpret
 (M,Oi,Os)
 (CFG2 inputInterpret certInterpret inputTest certs stateInterpret
       (x::ins) state outStream)) ==>
(TR2 (M,Oi,Os) (exec(CMD cmd))
 (CFG2 inputInterpret certInterpret inputTest certs stateInterpret
       (x::ins) state outStream)
 (CFG2 inputInterpret certInterpret inputTest certs stateInterpret
       ins (NS state (exec(CMD cmd))) ((Out state (exec(CMD cmd)))::outStream))))
/\
(!(inputInterpret: 'input -> ('command inst,'principal,'d,'e)Form)
 (certInterpret: 'cert -> ('command inst,'principal,'d,'e)Form)
 (inputTest:('command inst,'principal,'d,'e)Form -> bool)
 (x:'input)
 (NS: 'state -> 'command inst trType -> 'state)
 (M:('command inst,'b,'principal,'d,'e)Kripke)
 (Oi:'d po)
 (Os:'e po)
 (Out: 'state -> 'command inst trType -> 'output)
 (state:'state)
 (certs:'cert list)
 (stateInterpret:'state -> ('command inst,'principal,'d,'e)Form)
 (cmd:'command)
 (ins:'input list)
 (outStream:'output list).
(inputTest(inputInterpret (x:'input))) /\
(CFG2Interpret
 (M,Oi,Os)
 (CFG2 inputInterpret certInterpret inputTest certs stateInterpret
       (x::ins) state outStream)) ==>
(TR2 (M,Oi,Os) (trap(CMD cmd))
 (CFG2 inputInterpret certInterpret inputTest certs stateInterpret
       (x::ins) state outStream)
 (CFG2 inputInterpret certInterpret inputTest certs stateInterpret
       ins (NS state (trap(CMD cmd))) ((Out state (trap(CMD cmd)))::outStream))))
/\
(!(inputInterpret: 'input -> ('command inst,'principal,'d,'e)Form)
 (certInterpret: 'cert -> ('command inst,'principal,'d,'e)Form)
 (inputTest:('command inst,'principal,'d,'e)Form -> bool)
 (x:'input)
 (NS: 'state -> 'command inst trType -> 'state)
 (M:('command inst,'b,'principal,'d,'e)Kripke)
 (Oi:'d po)
 (Os:'e po)
 (Out: 'state -> 'command inst trType -> 'output)
 (state:'state)
 (certs:'cert list)
 (stateInterpret:'state -> ('command inst,'principal,'d,'e)Form)
 (cmd:'command)
 (ins:'input list)
 (outStream:'output list).
~(inputTest(inputInterpret (x:'input))) ==>
(TR2 (M,Oi,Os) discard
 (CFG2 inputInterpret certInterpret inputTest certs stateInterpret
       (x::ins) state outStream)
 (CFG2 inputInterpret certInterpret inputTest certs stateInterpret
       ins (NS state discard) ((Out state discard)::outStream))))`
```

D.2 Defining Properties of TR2

[TR2_rules]

$$
\begin{array}{l}
\vdash\ (\forall\, \mathit{inputInterpret}\ \mathit{certInterpret}\ \mathit{inputTest}\ x\ \mathit{NS}\ M\ \mathit{Oi}\ \mathit{Os}\ \mathit{Out}\\
\qquad \mathit{state}\ \mathit{certs}\ \mathit{stateInterpret}\ \mathit{cmd}\ \mathit{ins}\ \mathit{outStream}.\\
\qquad\ \mathit{inputTest}\ (\mathit{inputInterpret}\ x)\ \wedge\\
\qquad\ \texttt{CFG2Interpret}\ (M, \mathit{Oi}, \mathit{Os})\\
\qquad\quad (\texttt{CFG2}\ \mathit{inputInterpret}\ \mathit{certInterpret}\ \mathit{inputTest}\ \mathit{certs}\\
\qquad\qquad \mathit{stateInterpret}\ (x::\mathit{ins})\ \mathit{state}\ \mathit{outStream})\ \Rightarrow\\
\qquad\ \texttt{TR2}\ (M, \mathit{Oi}, \mathit{Os})\ (\texttt{exec}\ (\texttt{CMD}\ \mathit{cmd}))\\
\qquad\quad (\texttt{CFG2}\ \mathit{inputInterpret}\ \mathit{certInterpret}\ \mathit{inputTest}\ \mathit{certs}\\
\qquad\qquad \mathit{stateInterpret}\ (x::\mathit{ins})\ \mathit{state}\ \mathit{outStream})\\
\qquad\quad (\texttt{CFG2}\ \mathit{inputInterpret}\ \mathit{certInterpret}\ \mathit{inputTest}\ \mathit{certs}\\
\qquad\qquad \mathit{stateInterpret}\ \mathit{ins}\ (\mathit{NS}\ \mathit{state}\ (\texttt{exec}\ (\texttt{CMD}\ \mathit{cmd})))\\
\qquad\qquad (\mathit{Out}\ \mathit{state}\ (\texttt{exec}\ (\texttt{CMD}\ \mathit{cmd}))::\mathit{outStream})))\ \wedge\\
\quad (\forall\, \mathit{inputInterpret}\ \mathit{certInterpret}\ \mathit{inputTest}\ x\ \mathit{NS}\ M\ \mathit{Oi}\ \mathit{Os}\ \mathit{Out}\\
\qquad \mathit{state}\ \mathit{certs}\ \mathit{stateInterpret}\ \mathit{cmd}\ \mathit{ins}\ \mathit{outStream}.\\
\qquad\ \mathit{inputTest}\ (\mathit{inputInterpret}\ x)\ \wedge\\
\qquad\ \texttt{CFG2Interpret}\ (M, \mathit{Oi}, \mathit{Os})\\
\qquad\quad (\texttt{CFG2}\ \mathit{inputInterpret}\ \mathit{certInterpret}\ \mathit{inputTest}\ \mathit{certs}\\
\qquad\qquad \mathit{stateInterpret}\ (x::\mathit{ins})\ \mathit{state}\ \mathit{outStream})\ \Rightarrow\\
\qquad\ \texttt{TR2}\ (M, \mathit{Oi}, \mathit{Os})\ (\texttt{trap}\ (\texttt{CMD}\ \mathit{cmd}))\\
\qquad\quad (\texttt{CFG2}\ \mathit{inputInterpret}\ \mathit{certInterpret}\ \mathit{inputTest}\ \mathit{certs}\\
\qquad\qquad \mathit{stateInterpret}\ (x::\mathit{ins})\ \mathit{state}\ \mathit{outStream})\\
\qquad\quad (\texttt{CFG2}\ \mathit{inputInterpret}\ \mathit{certInterpret}\ \mathit{inputTest}\ \mathit{certs}\\
\qquad\qquad \mathit{stateInterpret}\ \mathit{ins}\ (\mathit{NS}\ \mathit{state}\ (\texttt{trap}\ (\texttt{CMD}\ \mathit{cmd})))\\
\qquad\qquad (\mathit{Out}\ \mathit{state}\ (\texttt{trap}\ (\texttt{CMD}\ \mathit{cmd}))::\mathit{outStream})))\ \wedge\\
\quad \forall\, \mathit{inputInterpret}\ \mathit{certInterpret}\ \mathit{inputTest}\ x\ \mathit{NS}\ M\ \mathit{Oi}\ \mathit{Os}\ \mathit{Out}\\
\qquad \mathit{state}\ \mathit{certs}\ \mathit{stateInterpret}\ \mathit{cmd}\ \mathit{ins}\ \mathit{outStream}.\\
\qquad \neg \mathit{inputTest}\ (\mathit{inputInterpret}\ x)\ \Rightarrow\\
\qquad\ \texttt{TR2}\ (M, \mathit{Oi}, \mathit{Os})\ \texttt{discard}\\
\qquad\quad (\texttt{CFG2}\ \mathit{inputInterpret}\ \mathit{certInterpret}\ \mathit{inputTest}\ \mathit{certs}\\
\qquad\qquad \mathit{stateInterpret}\ (x::\mathit{ins})\ \mathit{state}\ \mathit{outStream})\\
\qquad\quad (\texttt{CFG2}\ \mathit{inputInterpret}\ \mathit{certInterpret}\ \mathit{inputTest}\ \mathit{certs}\\
\qquad\qquad \mathit{stateInterpret}\ \mathit{ins}\ (\mathit{NS}\ \mathit{state}\ \texttt{discard})\\
\qquad\qquad (\mathit{Out}\ \mathit{state}\ \texttt{discard}::\mathit{outStream}))
\end{array}
$$

[TR2_ind]

$$
\begin{array}{l}
\vdash\ \forall TR_2'.\\
\qquad (\forall\, \mathit{inputInterpret}\ \mathit{certInterpret}\ \mathit{inputTest}\ x\ \mathit{NS}\ M\ \mathit{Oi}\ \mathit{Os}\ \mathit{Out}\\
\qquad\quad \mathit{state}\ \mathit{certs}\ \mathit{stateInterpret}\ \mathit{cmd}\ \mathit{ins}\ \mathit{outStream}.\\
\qquad\quad \mathit{inputTest}\ (\mathit{inputInterpret}\ x)\ \wedge\\
\qquad\quad \texttt{CFG2Interpret}\ (M, \mathit{Oi}, \mathit{Os})\\
\qquad\qquad (\texttt{CFG2}\ \mathit{inputInterpret}\ \mathit{certInterpret}\ \mathit{inputTest}\ \mathit{certs}\\
\qquad\qquad\quad \mathit{stateInterpret}\ (x::\mathit{ins})\ \mathit{state}\ \mathit{outStream})\ \Rightarrow\\
\qquad\quad TR_2'\ (M, \mathit{Oi}, \mathit{Os})\ (\texttt{exec}\ (\texttt{CMD}\ \mathit{cmd}))\\
\qquad\qquad (\texttt{CFG2}\ \mathit{inputInterpret}\ \mathit{certInterpret}\ \mathit{inputTest}\ \mathit{certs}\\
\qquad\qquad\quad \mathit{stateInterpret}\ (x::\mathit{ins})\ \mathit{state}\ \mathit{outStream})\\
\qquad\qquad (\texttt{CFG2}\ \mathit{inputInterpret}\ \mathit{certInterpret}\ \mathit{inputTest}\ \mathit{certs}\\
\qquad\qquad\quad \mathit{stateInterpret}\ \mathit{ins}\ (\mathit{NS}\ \mathit{state}\ (\texttt{exec}\ (\texttt{CMD}\ \mathit{cmd})))\\
\qquad\qquad\quad (\mathit{Out}\ \mathit{state}\ (\texttt{exec}\ (\texttt{CMD}\ \mathit{cmd}))::\mathit{outStream})))\ \wedge\\
\qquad (\forall\, \mathit{inputInterpret}\ \mathit{certInterpret}\ \mathit{inputTest}\ x\ \mathit{NS}\ M\ \mathit{Oi}\ \mathit{Os}\ \mathit{Out}\\
\qquad\quad \mathit{state}\ \mathit{certs}\ \mathit{stateInterpret}\ \mathit{cmd}\ \mathit{ins}\ \mathit{outStream}.\\
\qquad\quad \mathit{inputTest}\ (\mathit{inputInterpret}\ x)\ \wedge\\
\qquad\quad \texttt{CFG2Interpret}\ (M, \mathit{Oi}, \mathit{Os})\\
\qquad\qquad (\texttt{CFG2}\ \mathit{inputInterpret}\ \mathit{certInterpret}\ \mathit{inputTest}\ \mathit{certs}\\
\qquad\qquad\quad \mathit{stateInterpret}\ (x::\mathit{ins})\ \mathit{state}\ \mathit{outStream})\ \Rightarrow
\end{array}
$$

TR'_2 (M, Oi, Os) (trap (CMD cmd))
(CFG2 $inputInterpret$ $certInterpret$ $inputTest$ $certs$
$stateInterpret$ $(x::ins)$ $state$ $outStream$)
(CFG2 $inputInterpret$ $certInterpret$ $inputTest$ $certs$
$stateInterpret$ ins $(NS$ $state$ (trap (CMD cmd)))
$(Out$ $state$ (trap (CMD cmd))$::outStream$))) $\wedge$
$(\forall inputInterpret$ $certInterpret$ $inputTest$ x NS M Oi Os Out
$state$ $certs$ $stateInterpret$ cmd ins $outStream$.
$\neg inputTest$ $(inputInterpret$ $x)$ $\Rightarrow$
TR'_2 (M, Oi, Os) discard
(CFG2 $inputInterpret$ $certInterpret$ $inputTest$ $certs$
$stateInterpret$ $(x::ins)$ $state$ $outStream$)
(CFG2 $inputInterpret$ $certInterpret$ $inputTest$ $certs$
$stateInterpret$ ins $(NS$ $state$ discard)
$(Out$ $state$ discard$::outStream$))) $\Rightarrow$
$\forall a_0$ a_1 a_2 a_3. TR2 a_0 a_1 a_2 a_3 $\Rightarrow$ TR'_2 a_0 a_1 a_2 a_3

[TR2_cases]

$\vdash$ $\forall a_0$ a_1 a_2 a_3.
TR2 a_0 a_1 a_2 a_3 $\Longleftrightarrow$
$(\exists inputInterpret$ $certInterpret$ $inputTest$ x NS M Oi Os Out
$state$ $certs$ $stateInterpret$ cmd ins $outStream$.
$(a_0 = (M, Oi, Os))$ $\wedge$ $(a_1 =$ exec (CMD cmd)) $\wedge$
$(a_2 =$
CFG2 $inputInterpret$ $certInterpret$ $inputTest$ $certs$
$stateInterpret$ $(x::ins)$ $state$ $outStream$) $\wedge$
$(a_3 =$
CFG2 $inputInterpret$ $certInterpret$ $inputTest$ $certs$
$stateInterpret$ ins $(NS$ $state$ (exec (CMD cmd)))
$(Out$ $state$ (exec (CMD cmd))$::outStream$)) $\wedge$
$inputTest$ $(inputInterpret$ $x)$ $\wedge$
CFG2Interpret (M, Oi, Os)
(CFG2 $inputInterpret$ $certInterpret$ $inputTest$ $certs$
$stateInterpret$ $(x::ins)$ $state$ $outStream$)) $\vee$
$(\exists inputInterpret$ $certInterpret$ $inputTest$ x NS M Oi Os Out
$state$ $certs$ $stateInterpret$ cmd ins $outStream$.
$(a_0 = (M, Oi, Os))$ $\wedge$ $(a_1 =$ trap (CMD cmd)) $\wedge$
$(a_2 =$
CFG2 $inputInterpret$ $certInterpret$ $inputTest$ $certs$
$stateInterpret$ $(x::ins)$ $state$ $outStream$) $\wedge$
$(a_3 =$
CFG2 $inputInterpret$ $certInterpret$ $inputTest$ $certs$
$stateInterpret$ ins $(NS$ $state$ (trap (CMD cmd)))
$(Out$ $state$ (trap (CMD cmd))$::outStream$)) $\wedge$
$inputTest$ $(inputInterpret$ $x)$ $\wedge$
CFG2Interpret (M, Oi, Os)
(CFG2 $inputInterpret$ $certInterpret$ $inputTest$ $certs$
$stateInterpret$ $(x::ins)$ $state$ $outStream$)) $\vee$
$\exists inputInterpret$ $certInterpret$ $inputTest$ x NS M Oi Os Out
$state$ $certs$ $stateInterpret$ cmd ins $outStream$.
$(a_0 = (M, Oi, Os))$ $\wedge$ $(a_1 =$ discard) $\wedge$
$(a_2 =$
CFG2 $inputInterpret$ $certInterpret$ $inputTest$ $certs$
$stateInterpret$ $(x::ins)$ $state$ $outStream$) $\wedge$
$(a_3 =$
CFG2 $inputInterpret$ $certInterpret$ $inputTest$ $certs$
$stateInterpret$ ins $(NS$ $state$ discard)
$(Out$ $state$ discard$::outStream$)) $\wedge$

$\neg inputTest\ \ (inputInterpret\ \ x)$

Appendix E

HOL Definition of isAuthenticated

[isAuthenticated_def]

```
⊢ (isAuthenticated
      (Name Keyboard quoting Name (Owner ownerID) says
       prop (CMD cmd)) ⟺ T) ∧
   (isAuthenticated
      (Name (Key (pubK Server)) quoting
       Name (Owner ownerID) says prop (CMD cmd)) ⟺ T) ∧
   (isAuthenticated
      (Name (Key (pubK Server)) quoting
       Name (Role (Utility utilityID)) says prop (CMD cmd)) ⟺
    T) ∧ (isAuthenticated TT ⟺ F) ∧ (isAuthenticated FF ⟺ F) ∧
   (isAuthenticated (prop v) ⟺ F) ∧
   (isAuthenticated (notf v1) ⟺ F) ∧
   (isAuthenticated (v2 andf v3) ⟺ F) ∧
   (isAuthenticated (v4 orf v5) ⟺ F) ∧
   (isAuthenticated (v6 impf v7) ⟺ F) ∧
   (isAuthenticated (v8 eqf v9) ⟺ F) ∧
   (isAuthenticated (v10 says TT) ⟺ F) ∧
   (isAuthenticated (v10 says FF) ⟺ F) ∧
   (isAuthenticated (Name v132 says prop v66) ⟺ F) ∧
   (isAuthenticated (v133 meet v134 says prop v66) ⟺ F) ∧
   (isAuthenticated
      (Name (Role v174) quoting Name (Role v164) says
       prop (CMD v142)) ⟺ F) ∧
   (isAuthenticated
      (Name (Key v175) quoting Name (Role CA) says
       prop (CMD v142)) ⟺ F) ∧
   (isAuthenticated
      (Name (Key v175) quoting Name (Role Server) says
       prop (CMD v142)) ⟺ F) ∧
   (isAuthenticated
      (Name (Key (pubK CA)) quoting
       Name (Role (Utility v184)) says prop (CMD v142)) ⟺ F) ∧
   (isAuthenticated
      (Name (Key (pubK (Utility v190))) quoting
       Name (Role (Utility v184)) says prop (CMD v142)) ⟺ F) ∧
   (isAuthenticated
      (Name (Key (privK v187)) quoting
       Name (Role (Utility v184)) says prop (CMD v142)) ⟺ F) ∧
   (isAuthenticated
      (Name Keyboard quoting Name (Role v164) says
       prop (CMD v142)) ⟺ F) ∧
   (isAuthenticated
      (Name (Owner v176) quoting Name (Role v164) says
       prop (CMD v142)) ⟺ F) ∧
   (isAuthenticated
      (Name (Account v177 v178) quoting Name (Role v164) says
```

```
    prop (CMD v142)) ⟺ F) ∧
(isAuthenticated
   (Name v154 quoting Name (Key v165) says
    prop (CMD v142)) ⟺ F) ∧
(isAuthenticated
   (Name v154 quoting Name Keyboard says prop (CMD v142)) ⟺
 F) ∧
(isAuthenticated
   (Name (Role v192) quoting Name (Owner v166) says
    prop (CMD v142)) ⟺ F) ∧
(isAuthenticated
   (Name (Key (pubK CA)) quoting Name (Owner v166) says
    prop (CMD v142)) ⟺ F) ∧
(isAuthenticated
   (Name (Key (pubK (Utility v206))) quoting
    Name (Owner v166) says prop (CMD v142)) ⟺ F) ∧
(isAuthenticated
   (Name (Key (privK v203)) quoting Name (Owner v166) says
    prop (CMD v142)) ⟺ F) ∧
(isAuthenticated
   (Name (Owner v194) quoting Name (Owner v166) says
    prop (CMD v142)) ⟺ F) ∧
(isAuthenticated
   (Name (Account v195 v196) quoting Name (Owner v166) says
    prop (CMD v142)) ⟺ F) ∧
(isAuthenticated
   (Name v154 quoting Name (Account v167 v168) says
    prop (CMD v142)) ⟺ F) ∧
(isAuthenticated
   (v155 meet v156 quoting Name v144 says prop (CMD v142)) ⟺
 F) ∧
(isAuthenticated
   ((v157 quoting v158) quoting Name v144 says
    prop (CMD v142)) ⟺ F) ∧
(isAuthenticated
   (v135 quoting v145 meet v146 says prop (CMD v142)) ⟺ F) ∧
(isAuthenticated
   (v135 quoting v147 quoting v148 says prop (CMD v142)) ⟺
 F) ∧
(isAuthenticated (v135 quoting v136 says prop TRAP) ⟺ F) ∧
(isAuthenticated (v10 says notf v67) ⟺ F) ∧
(isAuthenticated (v10 says (v68 andf v69)) ⟺ F) ∧
(isAuthenticated (v10 says (v70 orf v71)) ⟺ F) ∧
(isAuthenticated (v10 says (v72 impf v73)) ⟺ F) ∧
(isAuthenticated (v10 says (v74 eqf v75)) ⟺ F) ∧
(isAuthenticated (v10 says v76 says v77) ⟺ F) ∧
(isAuthenticated (v10 says v78 speaks_for v79) ⟺ F) ∧
(isAuthenticated (v10 says v80 controls v81) ⟺ F) ∧
(isAuthenticated (v10 says reps v82 v83 v84) ⟺ F) ∧
(isAuthenticated (v10 says v85 domi v86) ⟺ F) ∧
(isAuthenticated (v10 says v87 eqi v88) ⟺ F) ∧
(isAuthenticated (v10 says v89 doms v90) ⟺ F) ∧
(isAuthenticated (v10 says v91 eqs v92) ⟺ F) ∧
(isAuthenticated (v10 says v93 eqn v94) ⟺ F) ∧
(isAuthenticated (v10 says v95 lte v96) ⟺ F) ∧
(isAuthenticated (v10 says v97 lt v98) ⟺ F) ∧
(isAuthenticated (v12 speaks_for v13) ⟺ F) ∧
(isAuthenticated (v14 controls v15) ⟺ F) ∧
```

(isAuthenticated (reps v_{16} v_{17} v_{18}) $\iff$ F) $\wedge$
(isAuthenticated (v_{19} domi v_{20}) $\iff$ F) $\wedge$
(isAuthenticated (v_{21} eqi v_{22}) $\iff$ F) $\wedge$
(isAuthenticated (v_{23} doms v_{24}) $\iff$ F) $\wedge$
(isAuthenticated (v_{25} eqs v_{26}) $\iff$ F) $\wedge$
(isAuthenticated (v_{27} eqn v_{28}) $\iff$ F) $\wedge$
(isAuthenticated (v_{29} lte v_{30}) $\iff$ F) $\wedge$
(isAuthenticated (v_{31} lt v_{32}) $\iff$ F)

Bibliography

[1] IEEE Guide for Information Technology–System Definition–Concept of Operations (ConOps) Document, 19 March 1998. IEEE Computer Society, IEEE Std 1362-1998.

[2] JP 5-0, Joint Operation Planning, 11 August 2011. US Dept of Defense.

[3] *Cyber-Assurance for the Internet of Things*. IEEE Press/Wiley, 2017.

[4] ABADI, M., BURROWS, M., LAMPSON, B., AND PLOTKIN, G. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems 15*, 4 (September 1993), 706–734.

[5] BELL, D. E., AND LA PADULA, L. J. Secure computer systems: Mathematical foundations. Tech. Rep. Technical Report MTR-2547, Vol. I, MITRE Corporation, Bedford, MA, March 1973.

[6] BELL, D. E., AND LA PADULA, L. J. Secure computer system: Unified exposition and Multics interpretation. Tech. Rep. MTR-2997 Rev. 1, MITRE Corporation, Bedford, MA, March 1975.

[7] BIBA, K. Integrity considerations for secure computer systems. Tech. Rep. MTR-3153, MITRE Corporation, Bedford, MA, June 1975.

[8] CHIN, S.-K., AND OLDER, S. *Access Control, Security, and Trust: A Logical Approach*. CRC Press/Taylor Francis, 2011.

[9] CONWAY, L. Reminiscences of the vlsi revolution: How a series of failures triggered a paradigm shift in digital design. *IEEE Solid-State Circuits Magazine 4*, 4 (Fall 2012), 8–31.

[10] FERRAIOLO, D., AND KUHN, R. Role-Based Access Control. In *15th NIST-NCSC National Computer Security Conference* (Gaithersburg, MD, 1992), pp. 554–563.

[11] GORDON, M., AND MELHAM, T. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, New York, 1993.

[12] POPEK, G. J., AND GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Communications of the ACM 17*, 7 (July 1974), 412–421.

[13] SANDHU, R. S., COYNE, E. J., FEINSTEIN, H. L., AND YOUMAN, C. E. Role-based access control models. *IEEE Computer 29*, 2 (February 1996), 38–47.