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 [8]

"... 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 $\lambda$ 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 se-
curity 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 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

   \[ \text{Alice says } \langle \text{command}_1 \rangle. \]

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 \textit{go}, then he is to issue another command, such as \textit{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

   \[ \text{Bob says } \langle \text{command}_2 \rangle. \]

4. What Figure 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 \textit{command}_2. This is symbolized by

   \[ \text{Bob says } \langle \text{command}_2 \rangle. \]
Regarding the comment in Figure 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 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.

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 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 [7] and is an extension and modification of an
access-control logic for distributed systems [3].

• Computer-assisted reasoning tools to (1) formally verify all proofs and assur-
ance 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 [10]. It is freely available and has been in use since 1987.

• A model of idealized cryptographic operations and their properties imple-
mented in HOL.

• Models of state machine transition systems incorporating authentication, au-
thorization, next-state functions, and output functions as parameters in sup-
port of security and to avoid state explosion. Our networked thermostat illus-
tration build upon the foundations of virtual machines, in particular [11].

1.3 Chapter Outline

The remainder of this chapter is organized as follows.

• Section 2 defines the syntax, semantics, and inference rules for an access-
control logic used to reason about command and control (C2).

• Section 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.

• Section 4 describes the HOL implementation of the access-control logic and
the C2 calculus.

• Section 5 describes algebraic models of ideal cryptographic operations such
as hashing, symmetric and asymmetric encryption, and cryptographic sign-
ning and verification. These algebraic models are implemented in HOL.

• Section 6 shows how security is built into state machines by labeled-transition
descriptions incorporating security policies described in the access-control
logic.

• Section 7 is a detailed example showing how security is designed into a
networked thermostat.
Section 8 contains our conclusions.

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 [7]. 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.

$$Princ ::= PName / Princ & Princ / Princ | Princ$$

“ & ” 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

$$Alice K_{Alice} Alice & Bob Alice | Bob$$
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 $⇒$ (pronounced “speaks for”), says, controls, and reps.

In this presentation of the C2 calculus, we use 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.

\[
Form ::= \text{PropVar} / \neg Form /
\]
\[
(Form ∨ Form) / (Form ∧ Form) /
\]
\[
(Form ⇒ Form) / (Form ≡ Form) /
\]
\[
(Princ ⇒ Princ) / (Princ says Form) /
\]
\[
(Princ controls Form) / Princ reps Princ on Form
\]

Figure 3 is a table of typical C2 statements and their representation as formulas in the C2 calculus.

### 2.2 Semantics

The semantics of the access-control logic uses Kripke structures. A Kripke structure $M$ is a three-tuple $(W,I,J)$, where:

- $W$ is a nonempty set, whose elements are called worlds.
• \( I : \text{PropVar} \rightarrow \mathcal{P}(W) \) is an interpretation function that maps each propositional variable \( p \) to a set of worlds.

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

\[
\hat{J}(A) = J(A) \\
\hat{J}(P \& Q) = \hat{J}(P) \cup \hat{J}(Q) \\
\hat{J}(P \mid Q) = \hat{J}(P) \circ \hat{J}(Q).
\]

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 \( M = \langle W, I, J \rangle \) gives rise to a semantic function \( E_M[\cdot] : \text{Form} \rightarrow \mathcal{P}(W) \), where \( E_M[\varphi] \) is the set of worlds in which \( \varphi \) is considered true.

\( E_M[\varphi] \) is defined inductively on the structure of \( \varphi \), as shown in Figure 4. Note, in the definition of \( E_M[P \text{ says } \varphi] \), that \( \hat{J}(P)(w) \) is simply the image of world \( w \) under the relation \( J(P) \).

### 2.3 Inference Rules

**Figure 5** Inference rules for the access-control logic

<table>
<thead>
<tr>
<th>Modus Ponens</th>
<th>( P \text{ controls } \varphi \stackrel{\text{def}}{=} (P \text{ says } \varphi \supset \varphi) )</th>
<th>( P \text{ says } \varphi )</th>
<th>( P \text{ reps } Q \text{ on } \varphi \stackrel{\text{def}}{=} P \mid Q \text{ says } \varphi \supset Q \text{ says } \varphi )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Says</strong></td>
<td>( \frac{\varphi \supset \varphi'}{\varphi'} )</td>
<td>( \frac{P \text{ says } \varphi}{\varphi} )</td>
<td>( \frac{P \text{ says } \varphi}{\varphi} )</td>
</tr>
<tr>
<td><strong>Repuses</strong></td>
<td>( \frac{Q \text{ controls } \varphi}{P \text{ reps } Q \text{ on } \varphi} )</td>
<td>( \frac{P \text{ says } \varphi}{P \mid Q \text{ says } \varphi} )</td>
<td>( \frac{Q \text{ says } \varphi}{P \mid Q \text{ says } \varphi} )</td>
</tr>
<tr>
<td><strong>Derived</strong></td>
<td>( \frac{P \Rightarrow Q}{Q \text{ says } \varphi} )</td>
<td>( \frac{Q \text{ says } \varphi}{P \mid Q \text{ says } \varphi} )</td>
<td>( \frac{Q \text{ says } \varphi}{P \mid Q \text{ says } \varphi} )</td>
</tr>
<tr>
<td><strong>Speaks For</strong></td>
<td>( \frac{P \Rightarrow Q \mid Q \text{ says } \varphi}{P \text{ says } \varphi} )</td>
<td>( \frac{Q \text{ says } \varphi}{P \mid Q \text{ says } \varphi} )</td>
<td>( \frac{Q \text{ says } \varphi}{P \mid Q \text{ says } \varphi} )</td>
</tr>
</tbody>
</table>

| **Quoting**   | \( \frac{P \mid Q \text{ says } \varphi}{P \text{ says } \varphi} \) | \( \frac{Q \text{ says } \varphi}{P \mid Q \text{ says } \varphi} \) | \( \frac{Q \text{ says } \varphi}{P \mid Q \text{ says } \varphi} \) |

| **Idempotency**| \( \frac{P \Rightarrow P}{P \Rightarrow P} \) | \( \frac{P \Rightarrow P \mid Q \Rightarrow Q}{P \mid Q \Rightarrow P} \) | \( \frac{P \Rightarrow P \mid Q \Rightarrow Q}{P \mid Q \Rightarrow P} \) |

| **Monotonicity**| \( \frac{P \Rightarrow P \mid Q \Rightarrow Q}{P \Rightarrow P} \) | \( \frac{P \Rightarrow P \mid Q \Rightarrow Q}{P \Rightarrow P} \) | \( \frac{P \Rightarrow P \mid Q \Rightarrow Q}{P \Rightarrow P} \) |

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
\[ H_1 \ldots H_k \quad \rightarrow \quad C, \]
where \( H_1 \ldots H_k \) 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 definition of satisfies in the access-control logic. A Kripke structure \( \mathcal{M} \) satisfies a formula \( \varphi \) when \( E_{\mathcal{M}}[\varphi] = W \), i.e., \( \varphi \) is true in all worlds \( W \) of \( \mathcal{M} \). We denote \( \mathcal{M} \) satisfies \( \varphi \) 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 \( H_1 \ldots H_k \), 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 [7] are proved to be logically sound. Figure 5 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 2.1**
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 \( \langle \text{access files} \rangle \).
2. \( K_A \) is Alice’s key: \( K_A \Rightarrow Alice \).
3. Alice has permission to access the files: \( Alice \) controls \( \langle \text{access files} \rangle \)

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$\rangle$ \hspace{1cm} Digitally signed request
2. $K_A \Rightarrow Alice$ \hspace{1cm} Key associated with Alice
3. Alice controls $\langle$access files$\rangle$ \hspace{1cm} Alice’s capability to access files
4. Alice says $\langle$access files$\rangle$ \hspace{1cm} 2, 1 Derived Speaks For
5. $\langle$access files$\rangle$ \hspace{1cm} 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 5. The derived inference rule is

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

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, 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.

$$
P \text{ reps } Q \text{ on } \varphi \overset{\text{def}}{=} P \mid Q \text{ says } \varphi \Rightarrow Q \text{ says } \varphi
$$

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

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

In a command and control application, if you believe (1) Bob is authorized on command $\varphi$, (2) Alice is Bob’s delegate or representative on a command $\varphi$, and (3) Alice says Bob says command $\varphi$, then you are justified to conclude the command $\varphi$ 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 2.2**

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.

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 in mind, we do the following analysis from Bob's perspective.

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

   \[ K_A | \text{Commander says \langle go \rangle} \]

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:

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

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

   \( \text{Commander controls \langle go \rangle}. \)

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

   \( \langle go \rangle \supset \langle \text{launch} \rangle \)

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 \( K_B | \text{Operator says \langle launch \rangle} \). The proof is as follows using the inference rules in Figure 5.
1. $K_A | \text{Commander says } \langle go \rangle$
   Input signed by $K_A$

2. $K_A \Rightarrow Alice$
   Trust assumption—$K_A$ is Alice’s key

3. $Alice \text{ reps Commander on } \langle go \rangle$
   Trust assumption—Alice is acting as a Commander when issuing a go command

4. $\text{Commander controls } \langle go \rangle$
   Trust assumption—Commanders have authority to issue a go command

5. $\langle go \rangle \supset \langle \text{launch} \rangle$
   Policy assumption—if $go$ is true then so is $\text{launch}$

6. $\text{Commander } \Rightarrow \text{Commander}$
   Idempotence of $\Rightarrow$

7. $K_A | \text{Commander } \Rightarrow \text{Alice } | \text{Commander}$
   2, 6 Monotonicity of $|$  

8. $Alice | \text{Commander says } \langle go \rangle$
   7, 1 Derived Speaks For

9. $\langle go \rangle$
   4, 3, 8 Reps

10. $\langle \text{launch} \rangle$
    9, 5 Modus Ponens

11. $K_B | \text{Operator says } \langle \text{launch} \rangle$
    10 Says

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

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

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 $K_B | \text{Operator says } \langle \text{launch} \rangle$ is justified, where $K_B$ is Bob’s key.

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.

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 [10]. 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\textsuperscript{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 3.1**
Suppose we wish to define state machines parametrically in terms of their state, input, and next-state transition functions, as shown in Figure 6. States and inputs are envisioned to be any type and each may have an infinite number of elements. The notation in Figure 6 is used in HOL. Terms and their types are represented in HOL by \texttt{hol_term : hol_type}, i.e., HOL terms followed by their types separated by a colon. For example \texttt{1 : num}, states that \texttt{1} is of type \texttt{num} in HOL.

\textsuperscript{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.
HOL supports polymorphism by using type variables. Type variables in HOL all have a leading prime symbol, ’. Figure 6 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 6 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. 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 of `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 $\forall$; $\Rightarrow$ is logical implication $\Rightarrow$; and $\\land$ is logical conjunction $\land$.

```
val (even_rules, even_induction, even_cases) = Hol_reln
   'even 0 \n   (ln. even n $\Rightarrow$ 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 $\Gamma$ is a set of predicate logic terms. What $\Gamma \vdash t$ states is when all the terms in $\Gamma$ are true, then $t$ must be true, too. If $\Gamma$ is empty, then we write $\vdash t$. In each of the following three theorems, $\Gamma$ is empty.

```
[even_rules]
$\vdash$ even 0 $\land$ $\forall n$. even $n$ $\Rightarrow$ even $(n + 2)$

[even_induction]
$\vdash$ $\forall$even',
   even' 0 $\land$ ($\forall n$. even' $n$ $\Rightarrow$ even' $(n + 2)$) $\Rightarrow$
   $\forall a_0$. even $a_0$ $\Rightarrow$ even' $a_0$

[even_cases]
```
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 6 by \( \xrightarrow{\tau} \), 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 \( x \).

```
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.

\begin{align*}
[\text{Trans\_rules}] \\
\vdash \forall NS\ s\ x.\ Trans\ x\ s\ (NS\ s\ x) \\
[\text{Trans\_ind}] \\
\vdash \forall Trans'.
    \\
      (\forall NS\ s\ x.\ Trans'\ x\ s\ (NS\ s\ x)) \Rightarrow
      \forall a_0\ a_1\ a_2.\ Trans\ a_0\ a_1\ a_2 \Rightarrow Trans'\ a_0\ a_1\ a_2 \\
[\text{Trans\_cases}] \\
\vdash \forall a_0\ a_1\ a_2.\ Trans\ a_0\ a_1\ a_2 \iff \exists NS.\ a_2 = NS\ a_1\ a_0
\end{align*}

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.  

\( \diamond \)
**Example 3.2**

In this example we define another way of looking at the behavior of state machines. Figure 7 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 $\text{NS}$.

We can define a transition relation $\text{TR} x$ similar to the relation $\text{Trans} x$ in Example 3.1, 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.

```haskell
val = Hol_datatype 'configuration = CFG '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.

$\text{configuration} = \text{CFG} (\text{'input list}) \text{'state} (\text{'output list})$

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*.

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

$[\forall a_0 \ a_1 \ a_2 \ a'_0 \ a'_1 \ a'_2 .
  (\text{CFG} \ a_0 \ a_1 \ a_2 = \text{CFG} \ a'_0 \ a'_1 \ a'_2) \iff
  (a_0 = a'_0) \land (a_1 = a'_1) \land (a_2 = a'_2)]$

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

---

**Figure 7** State Machine Behavior with Input and Output Streams

![State Machine Diagram](image)
1. For all next-state functions $NS$, output functions $Out$, inputs $x$, input streams $ins$, states $s$, and output streams $outs$, the predicate $TR$ is true for configurations $(CFG (x::ins) s outs)$ and $(CFG ins (NS s x) (Out s x::outs))$.

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

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

```haskell
val ( TR rules , TR ind , TR cases ) = 
Hol_reln
  "INS 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]
\[ \forall NS Out s x ins outs . \]
\[ TR x (CFG (x::ins) s outs) \]
\[ (CFG ins (NS s x) (Out s x::outs)) \]

[TR_ind]
\[ \forall TR' . \]
\[ (\forall NS Out s x ins outs . \)
\[ TR' x (CFG (x::ins) s outs) \]
\[ (CFG ins (NS s x) (Out s x::outs)) \implies \]
\[ \forall a_0 a_1 a_2 . TR a_0 a_1 a_2 \implies TR' a_0 a_1 a_2 \]

[TR_cases]
\[ \forall a_0 a_1 a_2 . \]
\[ TR a_0 a_1 a_2 \iff \]
\[ \exists NS Out s ins outs . \]
\[ (a_1 = CFG (a_0::ins) s outs \land \]
\[ (a_2 = CFG ins (NS s a_0) (Out s a_0::outs)) \]

As with similar definitions, $TR$ rules is the formalization of rule (1), $TR$ ind is a result of $TR$ 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 3.3

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]
\[\vdash \text{Trans } x \ s \ (\text{NS } s \ x) \Rightarrow \]
\[\text{TR } x \ (\text{CFG } (x:\text{ins}) \ s \ \text{outs}) \ (\text{CFG } ins \ (NS \ s \ x) \ (Out \ s \ x::\text{outs}))\]

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 \(([],''(\text{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.

\[
\text{set_goal}([],''(\text{Trans } (x:'input) (s:'state) (NS s x)) ==> (\text{TR } x (\text{CFG } (x::\text{ins} \ s \ \text{outs}) (\text{CFG } ins (NS s x) (Out s x::\text{outs})))''));
\]

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.

\[
\text{e(STRIP_TAC)};
\]

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.

\[
\text{e(PROVE_TAC[TR_rules]);}
\]

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

\[
[\text{TR_Trans_lemma}]
\vdash \text{TR } x \ (\text{CFG } (x::\text{ins} \ s \ \text{outs}) \ (\text{CFG } ins (NS s x) \ (Out s x::\text{outs}) \Rightarrow}
With the two lemmas \textit{Trans_TR_lemma} and \textit{TR_Trans_lemma}, it is straightforward to prove that \textit{Trans} and \textit{TR} are logically equivalent. The following code snippet illustrates how the HOL function \texttt{TAC_PROOF} is used to prove the logical equivalence of \textit{Trans} and \textit{TR}.

```haskell
val Trans_Equiv_TR = TAC_PROOF
(([] , "(\(x : 'i n p u t\)
 (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.

```haskell
- val Trans_Equiv_TR = TAC_PROOF(
 ([] , "(\(x : 'i n p u t\)
 (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 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.

\textbf{Note:} in everything that follows, all formulas starting with \(\vdash\) are theorems in HOL, typeset in \LaTeX{} by HOL, and formally verified in HOL.

## 4 The Access-Control Logic in HOL

The access-control logic described in Section 2 is implemented in HOL by (1) defining its syntax as an algebraic type \textit{Form}, (2) inductively defining the semantic function \(E_M[[-]]\) in HOL over the type \textit{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,
Figure 8 Access-Control Logic Syntax in HOL

\begin{verbatim}
Form =
  TT
  | FF
  | prop 'aavar
  | notf ('aavar, 'apn, 'il, 'sl) Form
  | (andf) ('aavar, 'apn, 'il, 'sl) Form
  | (orf) ('aavar, 'apn, 'il, 'sl) Form
  | (impf) ('aavar, 'apn, 'il, 'sl) Form
  | (eqf) ('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)
  | (domi) ('apn, 'il) IntLevel
  | (eqi) ('apn, 'il) IntLevel
  | (eqs) ('apn, 'sl) SecLevel
  | (eqn) num num
  | (lte) num num
  | (lt) num num
\end{verbatim}

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 \textit{Form} algebraic type corresponding to access-control logic formulas, the algebraic type \textit{Princ} corresponding to principal expressions, and the algebraic type \textit{Kripke} corresponding to Kripke structures. The semantics of \textit{Form} and \textit{Princ} are defined using \textit{Kripke} and existing HOL operators. The properties of the access-control logic are proved as theorems in HOL.
Figure 9 Syntax of Principal Expressions, Integrity and Security Labels, and Kripke Structures in HOL

\begin{itemize}
\item \textbf{Princ} =
\begin{itemize}
\item \text{Name 'apn}
\item \text{(meet) ('apn Princ) ('apn Princ)}
\item \text{(quoting) ('apn Princ) ('apn Princ)}
\end{itemize}
\end{itemize}

\begin{itemize}
\item \textbf{IntLevel} = \text{iLab 'il | il 'apn ;}
\end{itemize}

\begin{itemize}
\item \textbf{SecLevel} = \text{sLab 'sl | sl 'apn}
\end{itemize}

\begin{itemize}
\item \textbf{Kripke} =
\begin{itemize}
\item \text{KS ('aavar -> 'aaworld -> bool)}
\item \text{('apn -> 'aaworld -> 'aaworld -> bool) ('apn -> 'il)}
\item \text{('apn -> 'sl)}
\end{itemize}
\end{itemize}

Figure 10 C2 Formulas and Their Representation in HOL

<table>
<thead>
<tr>
<th>C2 Formula</th>
<th>HOL Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨jump⟩</td>
<td>prop jump</td>
</tr>
<tr>
<td>¬(jump)</td>
<td>notf(prop jump)</td>
</tr>
<tr>
<td>⟨run⟩∧⟨jump⟩</td>
<td>prop run andf prop jump</td>
</tr>
<tr>
<td>⟨run⟩∨⟨stop⟩</td>
<td>prop run orf prop stop</td>
</tr>
<tr>
<td>⟨walk⟩≡⟨stop⟩</td>
<td>prop walk eqf prop stop</td>
</tr>
<tr>
<td>Alice says ⟨jump⟩</td>
<td>Name Alice says prop jump</td>
</tr>
<tr>
<td>Alice &amp; Bob says ⟨stop⟩</td>
<td>Name Alice meet Name Bob says prop stop</td>
</tr>
<tr>
<td>Bob</td>
<td>Carol says ⟨run⟩</td>
</tr>
<tr>
<td>Bob controls ⟨walk⟩</td>
<td>Name Bob controls prop walk</td>
</tr>
<tr>
<td>Bob reps Alice on ⟨jump⟩</td>
<td>reps(Name Bob)(Name Alice)(prop jump)</td>
</tr>
<tr>
<td>Carol ⇒ Bob</td>
<td>Name Carol speaks_for Name Bob</td>
</tr>
</tbody>
</table>

Figure 8 shows the HOL type \texttt{Form} corresponding to access-control logic formulas in HOL. Notice that the HOL implementation uses \texttt{notf}, \texttt{andf}, \texttt{orf}, \texttt{impf} and \texttt{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 \(M\). 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 8 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 \texttt{prop} and are applied to any type, as represented by \texttt{'aavar}. For example, \texttt{prop command} takes elements of the type \texttt{command} and maps them to propositions in the access-control logic in HOL.
Figure 9 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 [7].

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 10 is a table showing how formulas in the C2 calculus are written in HOL implementation of the access-control logic. The proposition 〈jump〉 is written as `prop jump` in HOL. Negation of a C2 formula, such as ¬〈jump〉 is written as `notf (prop jump)` in HOL. Alice says 〈jump〉 is written as `Name Alice says prop jump`, etc.

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 `E_M[¬ϕ]` in Figure 4, which defines the Kripke semantics of the access-control logic. The definition of `Efn` is in Section A.2. The definitions of `E_M[¬ϕ]` 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 4 and as described in [7]? The answer is if we can prove theorems in HOL about the HOL implementation that correspond to the inference rules in [7], then we are satisfied.

4.3 C2 Inference Rules in HOL

Recall in Section 2.3 that `M |= ϕ` denoted `E_M[ϕ] = W`, i.e., ϕ is true for all worlds in `M`. Inference rules in the C2 calculus are sound because whenever `M` satisfies all the hypotheses `H_1· · ·H_k`, then `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 A.2, 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 ]
\vdash \forall M O_i O_s f. (M, O_i, O_s) sat f \iff (Efn O_i O_s M f = U(:'world))
```

An inference rule in the C2 calculus of the form

```
H_1 · · ·H_k
\hline
C
```
Figure 11 HOL Theorems Corresponding to C2 Calculus Inference Rules (1 of 2)

[Controls_Eq]
\[ \vdash \forall M \, O_i \, O_s \, P \, f. \quad (M, O_i, O_s) \text{ sat } P \text{ controls } f \iff (M, O_i, O_s) \text{ sat } P \text{ says } f \text{ impf } f \]

[Reps_Eq]
\[ \vdash \forall M \, O_i \, O_s \, P \, Q \, f. \quad (M, O_i, O_s) \text{ sat } \text{reps } P \, Q \, f \iff (M, O_i, O_s) \text{ sat } P \text{ quoting } Q \text{ says } f \text{ impf } Q \text{ says } f \]

[Modus_Ponenens]
\[ \vdash \forall M \, O_i \, O_s \, f_1 \, f_2. \quad (M, O_i, O_s) \text{ sat } f_1 \Rightarrow (M, O_i, O_s) \text{ sat } f_1 \text{ impf } f_2 \Rightarrow (M, O_i, O_s) \text{ sat } f_2 \]

[Says]
\[ \vdash \forall M \, O_i \, O_s \, P \, f. \quad (M, O_i, O_s) \text{ sat } f \Rightarrow (M, O_i, O_s) \text{ sat } P \text{ says } f \]

[Controls]
\[ \vdash \forall M \, O_i \, O_s \, P \, f. \quad (M, O_i, O_s) \text{ sat } P \text{ says } f \Rightarrow (M, O_i, O_s) \text{ sat } P \text{ controls } f \Rightarrow (M, O_i, O_s) \text{ sat } f \]

[Derived_Speaks_For]
\[ \vdash \forall M \, O_i \, O_s \, P \, Q \, f. \quad (M, O_i, O_s) \text{ sat } P \text{ speaks_for } Q \Rightarrow (M, O_i, O_s) \text{ sat } P \text{ says } f \Rightarrow (M, O_i, O_s) \text{ sat } Q \text{ says } f \]

has a corresponding theorem in HOL

\[ \vdash \forall M \, O_i \, O_s \, (M, O_i, O_s) \text{ sat } H_1 \Rightarrow \cdots \Rightarrow (M, O_i, O_s) \text{ sat } H_k \Rightarrow (M, O_i, O_s) \text{ sat } C, \]

where \( \Rightarrow \) corresponds to logical implication in HOL. Figures 11 and 12 show the HOL theorems corresponding to the C2 inference rules in Figure 5.

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 [8]. 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 13 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.
5.1.1 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.

5.1.2 Modeling Idealized Behavior in HOL

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 14 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 15 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
Figure 15 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 a'0 a'1. (Es a0 a1 = Es a'0 a'1) ⇔ (a0 = a'0) ∧ (a1 = a'1)`

`[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))`

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.

- 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 16 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 17 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
Figure 18 Definitions and Properties of Asymmetric Keys and Messages

\[
pKey = \text{pubK 'princ} \mid \text{privK 'princ}
\]

[pKey_distinct_clauses]
\[\vdash (\forall a'. \text{pubK } a \neq \text{privK } a') \land \forall a'. \text{privK } a' \neq \text{pubK } a\]

[pKey_one_one]
\[\vdash (\forall a' \cdot (\text{pubK } a = \text{pubK } a') \iff (a = a')) \land \forall a' \cdot (\text{privK } a = \text{privK } a') \iff (a = a')\]

\[
\text{asymMsg} = E_a (\text{'princ pKey}) (\text{'message option})
\]

[asymMsg_one_one]
\[\vdash \forall a_0 a_1 a'_0 a'_1. (E_a a_0 a_1 = E_a a'_0 a'_1) \iff (a_0 = a'_0) \land (a_1 = a'_1)\]

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 \(K_{\text{Bob}}\). Only Bob, who alone possesses the private key \(K^{-1}_{\text{Bob}}\), is able to decrypt the message encrypted with his public key \(K_{\text{Bob}}\).

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 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 18 shows type definitions for asymmetric keys \(pKey\), i.e., public and private keys, and asymmetrically encrypted messages \(\text{asymMsg}\). Figure 18 also shows properties of \(pKey\) and \(\text{asymMsg}\).

- The type \(pKey\) has two forms, \(\text{pubK } P\) and \(\text{privK } P\), 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 \(\text{asymMsg}\) represents asymmetrically encrypted messages. The parameters of type constructor \(E_a\) are a \(pKey\) and a 'message option. Abstractly, the type constructor \(E_a\) stands for any asymmetric-key algorithm, e.g., RSA.
- Two \(\text{asymMsgs}\) are the same if they have the same \(pKey\) and 'message option values.
Figure 19: Definitions and Properties of Asymmetric Decryption

\[ \text{deciphP}_\text{def} \]
\[ \vdash (\text{deciphP} \ key \ (\text{Ea} \ (\text{privK} \ P) \ (\text{SOME} \ x)) = \\
\quad \text{if} \ key = \text{pubK} \ P \ \text{then} \ \text{SOME} \ x \ \text{else} \ \text{NONE}) \ \land \\
\quad (\text{deciphP} \ key \ (\text{Ea} \ (\text{pubK} \ P) \ (\text{SOME} \ x)) = \\
\quad \text{if} \ key = \text{privK} \ P \ \text{then} \ \text{SOME} \ x \ \text{else} \ \text{NONE}) \ \land \\
\quad (\text{deciphP} \ k_1 \ (\text{Ea} \ k_2 \ \text{NONE}) = \text{NONE}) \]  

\[ \text{deciphP}_\text{clauses} \]
\[ \vdash (\forall P \ text. \ (\text{deciphP} \ (\text{pubK} \ P) \ (\text{Ea} \ (\text{privK} \ P) \ (\text{SOME} \ text)) = \\
\text{SOME} \ text) \ \land \\
\quad (\text{deciphP} \ (\text{privK} \ P) \ (\text{Ea} \ (\text{pubK} \ P) \ (\text{SOME} \ text)) = \\
\text{SOME} \ text)) \ \land \\
\quad (\forall k \ P \ text. \ (\text{deciphP} \ k \ (\text{Ea} \ (\text{privK} \ P) \ (\text{SOME} \ text)) = \text{SOME} \ text) \ \iff \\
\quad (k = \text{pubK} \ P)) \ \land \\
\quad (\forall k \ P \ text. \ (\text{deciphP} \ k \ (\text{Ea} \ (\text{pubK} \ P) \ (\text{SOME} \ text)) = \text{SOME} \ text) \ \iff \\
\quad (k = \text{privK} \ P)) \ \land \\
\quad (\forall x \ k_1 \ k_2 \ P_1 \ P_2. \ (\text{deciphP} \ (\text{pubK} \ P_1) \ (\text{Ea} \ (\text{pubK} \ P_2) \ (\text{SOME} \ x)) = \text{NONE}) \ \land \\
\quad (\text{deciphP} \ k_1 \ (\text{Ea} \ k_2 \ \text{NONE}) = \text{NONE})) \ \land \\
\quad \forall x \ P_2 \ P_1. \ \text{deciphP} \ (\text{privK} \ P_1) \ (\text{Ea} \ (\text{privK} \ P_2) \ (\text{SOME} \ x)) = \text{NONE} \)  

Figure 19 shows the definition and properties of \text{deciphP}, which models the decryption of asymmetrically encrypted messages. Similar to symmetric-key encryption, to retrieve the plaintext \text{SOME} \ x requires use of the correct key, in this case \text{privK} \ P if the message was encrypted using \text{pubK} \ P, or \text{pubK} \ P if the message was encrypted with \text{privK} \ P. As before, garbage in produces garbage out.

The properties of \text{deciphP} are shown in Figures 19 and 20 by theorems \text{deciphP}_\text{clauses} and \text{deciphP}_\text{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 21, 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
Figure 20 One-to-One Properties of Asymmetric Decryption

\[ \vdash (\forall P_1, P_2, text_1, text_2. \ (deciphP (pubK P_1) (Ea (privK P_2) (SOME text_2)) = SOME text_1) \iff (P_1 = P_2) \land (text_1 = text_2)) \land \]

\[ (\forall P_1, P_2, text_1, text_2. \ (deciphP (privK P_1) (Ea (pubK P_2) (SOME text_2)) = SOME text_1) \iff (P_1 = P_2) \land (text_1 = text_2)) \land \]

\[ (\forall p, c, P, msg. \ (deciphP (pubK P) (Ea p c) = SOME msg) \iff (p = privK P) \land (c = SOME msg)) \land \]

\[ (\forall enMsg, P, msg. \ (deciphP (pubK P) enMsg = SOME msg) \iff (enMsg = Ea (privK P) (SOME msg))) \land \]

\[ (\forall p, c, P, msg. \ (deciphP (privK P) (Ea p c) = SOME msg) \iff (p = pubK P) \land (c = SOME msg)) \land \]

\[ (\forall enMsg, P, msg. \ (deciphP (privK P) enMsg = SOME msg) \iff (enMsg = Ea (pubK P) (SOME msg))) \]

Figure 21 Digital Signature Generation

![Diagram of digital signature generation](image)

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 22 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 23 shows the function definitions in HOL of \textit{sign} and \textit{signVerify}. \textit{sign} takes as inputs a \textit{pKey} and a digest and returns an asymmetrically encrypted digest using the asymmetric \textit{pKey}. \textit{signVerify} takes as input a \textit{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 \textit{signVerify} and \textit{sign} are in theorems \textit{signVerifyOK} and \textit{signVerify_one_one}.
• `signVerify` is always true for signatures generated as shown in Figure 21.

• `signVerify` and `sign` combine to have the desired properties that the plaintext must match and the corresponding keys must match.

6 Adding Security to State Machines

In this section, we use the infrastructure we have described in previous sections to add authentication andauthorization 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 Section 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 [5] and [4], the integrity policies of Biba [6], and role-based access control [9] and [12]. For illustrative purposes, the security policy we use is based on Popek and Goldberg’s virtualization policies [11]. 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

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

4. In keeping with the requirements for virtualizability as defined in [11], 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 24 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`. 
Figure 24 Definitions and Properties of Instruction and Transition Types

\[ \text{inst} = \text{CMD} \mid \text{TRAP} \]

[\text{inst\_distinct\_clauses}]
\[ \vdash (\forall a. \text{CMD} a \neq \text{TRAP}) \land (\forall a. \text{TRAP} \neq \text{CMD} a) \]

\[ \text{trType} = \text{discard} \mid \text{trap} \mid \text{exec} \]

[\text{trType\_distinct\_clauses}]
\[ \vdash (\forall a. \text{discard} \neq \text{trap} a) \land (\forall a. \text{discard} \neq \text{exec} a) \land \\
(\forall a'. a. \text{trap} a \neq \text{exec} a') \land (\forall a. \text{trap} a \neq \text{discard}) \land (\forall a'. a. \text{exec} a' \neq \text{trap} a) \]

Figure 25 Definition and Properties of High-Level SSM Configurations

\[ \text{configuration} = \]
\[ \text{CFG} ((\text{command inst, principal, d, e} \text{Form} \rightarrow \text{bool}) \\
(\text{state} \rightarrow (\text{command inst, principal, d, e} \text{Form}) \\
((\text{command inst, principal, d, e} \text{Form list}) \\
((\text{command inst, principal, d, e} \text{Form list}) \text{ state} \\
(\text{output list}) \\
[\text{configuration\_11}] \\
\vdash \forall a_0 a_1 a_2 a_3 a_4 a_5 a'_0 a'_1 a'_2 a'_3 a'_4 a'_5. \\
(\text{CFG} a_0 a_1 a_2 a_3 a_4 a_5 = \text{CFG} a'_0 a'_1 a'_2 a'_3 a'_4 a'_5) \iff \\
(a_0 = a'_0) \land (a_1 = a'_1) \land (a_2 = a'_2) \land (a_3 = a'_3) \land \\
(a_4 = a'_4) \land (a_5 = a'_5) \]

1. The purpose of \text{inst} is to add \text{TRAP} to the set of commands. Doing so facilitates writing policies in the access-control logic specifying when \text{TRAPs} should occur.

2. We can achieve the same effect by using option types, i.e., using \text{SOME} and \text{NONE}. To enhance readability, we use \text{CMD} and \text{TRAP} instead.

6.2 High-Level Secure State-Machine Description

Configurations Defined Figure 25 shows the definition of high-level SSM configurations and their properties. Configurations have six components.

1. An authentication function with type \((\text{command inst, principal, d, e})\text{Form} \rightarrow \text{bool}\) that returns \text{true} or \text{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 \(\text{state} \rightarrow (\text{command inst, principal, d, e})\text{Form}\) 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.
Figure 26 satList Definition and Properties

\[satList\_def\]
\[\forall M \, O_i \, O_s \, \text{formList}. \]
\((M, O_i, O_s) \text{ satList } \text{formList} \iff \]
\(\text{FOLDR} \ (\lambda x \, y. \ x \land y) \ T \ (\text{MAP} \ (\lambda f. \ (M, O_i, O_s) \text{ sat } f) \ \text{formList})\]

\[satList\_nil\]
\[\forall h \, t \, M \, O_i \, O_s. \]
\((M, O_i, O_s) \text{ satList } (h:t) \iff (M, O_i, O_s) \text{ sat } h \land (M, O_i, O_s) \text{ satList } t\]

\[satList\_CONJ\]
\[\forall l_1 \, l_2 \, M \, O_i \, O_s. \]
\((M, O_i, O_s) \text{ satList } l_1 \land (M, O_i, O_s) \text{ satList } l_2 \iff (M, O_i, O_s) \text{ satList } (l_1 ++ l_2)\]

Figure 27 Definition of Configuration Interpretation Function

\[CFGInterpret\_def\]
\[\forall M \, O_i \, O_s. \]
\((M, O_i, O_s) \text{ CFGInterpret } \text{inputTest} \text{ stateInterp} \text{ context} \ (x::ins) \text{ state} \text{outStream}) \iff \\
\((M, O_i, O_s) \text{ sat List context} \land (M, O_i, O_s) \text{ sat x} \land \\
(M, O_i, O_s) \text{ sat stateInterp state})\]

3. A list of access-control logic formulas \((\text{command inst, principal, d, e})\) 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.

4. An input stream of access-control logic formulas \((\text{command inst, principal, d, e})\) Form list.

5. The current state \(\text{state}\).

6. An output stream \(\text{output list}\).

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 26 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-
Figure 28 Discard Command Rule for Transition Relation TR

[TR_discard_cmd_rule]
\[
\vdash TR \ (M, O_i, O_s) \ discard \\
(CFG \ inputTest \ stateInterp \ (x::ins) \ s \ outs) \\
(CFG \ inputTest \ stateInterp \ certs \ ins \ (NS \ s \ discard) \\
(Out \ s \ discard::outs)) \iff \ \neg inputTest \ x
\]

Figure 29 Execute Command Rule for Transition Relation TR

[TR_exec_cmd_rule]
\[
\vdash \forall inputTest \ certs \ stateInterp \ P \ cmd \ ins \ s \ outs. \\
(\forall M \ O_i \ O_s. \\
\quad CFGInterpret \ (M, O_i, O_s) \\
\quad (CFG \ inputTest \ stateInterp \ certs \\
\quad (P \ says \ prop \ (CMD \ cmd)::ins) \ s \ outs) \Rightarrow \\
\quad (M, O_i, O_s) \ sat \ prop \ (CMD \ cmd)) \Rightarrow \\
\quad \forall NS \ Out \ M \ O_i \ O_s. \\
\quad TR \ (M, O_i, O_s) \ (exec \ (CMD \ cmd)) \\
\quad (CFG \ inputTest \ stateInterp \ certs \\
\quad (P \ says \ prop \ (CMD \ cmd)::ins) \ s \ outs) \\
\quad (CFG \ inputTest \ stateInterp \ certs \ ins \\
\quad (NS \ s \ (exec \ (CMD \ cmd))) \\
\quad (Out \ s \ (exec \ (CMD \ cmd)::outs)) \iff \\
\quad inputTest \ (P \ says \ prop \ (CMD \ cmd)) \land \\
\quad CFGInterpret \ (M, O_i, O_s) \\
\quad (CFG \ inputTest \ stateInterp \ certs \\
\quad (P \ says \ prop \ (CMD \ cmd)::ins) \ s \ outs) \land \\
\quad (M, O_i, O_s) \ sat \ prop \ (CMD \ cmd)
\]

reduction of \((M, O_i, O_s) \ sat\) mapped over each formula \(f_i\). For example,
\[
(M, O_i, O_s) \ sat \ List \ [f_1; f_2; \cdots; f_n] = (M, O_i, O_s) \ sat \ f_1 \land \cdots \land (M, O_i, O_s) \ sat \ f_n
\]

Interpretation of Configurations Defined Figure 27 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 3.2. This time, we account for the security interpretation of configurations.

Appendix A.3.1 gives the HOL source code for defining TR. Appendix A.3.2 shows the three defining properties of \(TR\) in HOL resulting from the inductive definition.
These properties are \textit{TR\_rules}, \textit{TR\_ind}, and \textit{TR\_cases}, which give the transition rules, induction property, and cases theorem, respectively.

Looking at \textit{TR\_rules}, we see there are three clauses, one each for the three \textit{trType}s labeling the transition relation \textit{TR} \((M, Oi, Os)\):

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

2. \textit{TR} \((M, Oi, Os)\) \textit{(trap (CMD cmd))}: the rule specifying when a command \textit{cmd} is trapped. The conditions are:
   
   (a) the input \textit{P says prop (CMD cmd)} must be authenticated by \textit{inputTest}, and
   
   (b) the security interpretation of the current configuration is given by \textit{CFGInterpret}.

3. \textit{TR} \((M, Oi, Os)\) \textit{discard}: the rule specifying when an input \textit{x} is discarded. The rule states when \textit{x} fails to be authenticated by \textit{inputTest}, \textit{x} is discarded from the input stream.

Based on the definitions of \textit{TR}, \textit{satList}, and \textit{CFGInterpret}, and their properties, we can prove three equality properties related to each of the transition types \textit{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. **TRdiscard_cmd_rule** as shown in Figure 28. It states that a discard transition occurs for an input \( x \) if and only if \( x \) fails to be authenticated, i.e., \( \neg \text{inputTest } x \) is true.

2. **TRexec_cmd_rule** as shown in Figure 29. It states that if \((M, O_t, O_s)\) sat \text{prop} \( \text{CMD} \) \( \text{cmd} \) is justified, i.e., implied by the security interpretation of the current configuration, as specified by \( \text{CFGInterpret} \), then \( \text{cmd} \) is executed if and only if (a) the input is authenticated, (b) \( \text{CFGInterpret} \) is the security interpretation, and (c) \((M, O_t, O_s)\) sat \text{prop} \( \text{CMD} \) \( \text{cmd} \) is true.

3. **TRtrap_cmd_rule** as shown in Figure 30. It states that if \((M, O_t, O_s)\) sat \text{prop} TRAP is justified, i.e., implied by the security interpretation of the current configuration, as specified by \( \text{CFGInterpret} \), then \( \text{cmd} \) is trapped if and only if (a) the input is authenticated, (b) \( \text{CFGInterpret} \) is the security interpretation, and (c) \((M, O_t, O_s)\) sat \text{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. \( \text{inputTest} \): the authentication function,
2. \( \text{stateInterp} \): the state interpretation function,
3. \( \text{certs} \): the credentials, trust assumptions, delegations, and authorizations informing authorization decisions,
4. \( \text{commands} \): commands are polymorphic,
5. \( \text{states} \): states are polymorphic,
6. \( \text{outputs} \): outputs are polymorphic,
7. \( \text{NS} \): the next-state function, and
8. \( \text{Out} \): the output function.

The three theorems in Figures 28, 29, and 30, 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 **TRdiscard_cmd_rule** the authentication function \( \text{inputTest} \) eliminates all unauthenticated commands.
2. In **TRexec_cmd_rule**, the condition
**Figure 31** Definition and Properties of Refined SSM Configurations and Interpretation

\[
\text{configuration}_2 = \\
\text{CFG}_2 \text{ ('input} \rightarrow \text{ ('command inst, 'principal, 'd, 'e) Form)} \\
\text{ ('cert} \rightarrow \text{ ('command inst, 'principal, 'd, 'e) Form)} \\
\text{ (('command inst, 'principal, 'd, 'e) Form} \rightarrow \text{ bool)} \\
\text{ ('cert list) } \\
\text{ ('state} \rightarrow \text{ ('command inst, 'principal, 'd, 'e) Form)} \\
\text{ ('input list) \ 'state \ ('output list)}
\]

[CFG2Interpret_def]
\[
\vdash \text{CFG2Interpret } (M, O_i, O_s) \\
\text{ (CFG2 inputInterpret certInterpret inputTest certs} \\
\text{ stateInterpret } (x::ins) \text{ state outStream) } \iff \\
\text{ (M, O_i, O_s) satList MAP certInterpret certs } \land \\
\text{ (M, O_i, O_s) sat inputInterpret x } \land \\
\text{ (M, O_i, O_s) sat stateInterpret state}
\]

\[
\forall M \ O_i \ O_s. \\
\text{CFGInterpret } (M, O_i, O_s) \\
\text{ (CFG inputTest stateInterp certs (P says prop (CMD cmd)::ins) s outs) } \Rightarrow \\
\text{ (M, O_i, O_s) sat prop (CMD cmd)}
\]


\[
\forall M \ O_i \ O_s. \\
\text{CFGInterpret } (M, O_i, O_s) \\
\text{ (CFG inputTest stateInterp certs (P says prop (CMD cmd)::ins) s outs) } \Rightarrow \\
\text{ (M, O_i, O_s) sat prop TRAP}
\]


3. In TR\_trap\_cmd\_rule, similar to TR\_exec\_cmd\_rule, the condition

\[
\forall M \ O_i \ O_s. \\
\text{CFGInterpret } (M, O_i, O_s) \\
\text{ (CFG inputTest stateInterp certs (P says prop (CMD cmd)::ins) s outs) } \Rightarrow \\
\text{ (M, O_i, O_s) sat prop TRAP}
\]

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.

### 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.
Figure 32 Discard Command Rule for Transition Relation TR2

\[ \text{TR2} \left( M, O_i, O_s \right) \text{ discard} \]

\[
\begin{align*}
\vdash \text{TR2} & \left( \text{CFG2 inputInterpret certInterpret inputTest certs stateInterpret (x::ins) state outStream} \right) \\
& \left( \text{CFG2 inputInterpret certInterpret inputTest certs stateInterpret ins (NS state discard)} \right) \\
& \left( \text{Out state discard::outStream} \right) \\
\iff \\
& \neg \text{inputTest (inputInterpret x)} 
\end{align*}
\]

Configurations and Their Interpretations Defined  Figure 31 shows the type definition of configuration2 and its interpretation function \( \text{CFG2Interpret} \). The refined configuration configuration2 has eight components.

1. An input interpretation function with type \( \text{input -> (command inst, principal, d, e)Form} \). This function gives meaning to inputs in the access-control logic.
2. A certificate interpretation function with type \( \text{cert -> (command inst, principal, d, e)Form} \). This function gives meaning to certificates in the access-control logic.
3. An authentication function with type \( \text{(command inst, principal, d, e)Form -> bool} \) that returns \text{true} or \text{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 \( \text{cert list} \) 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 \( \text{state -> (command inst, principal, d, e)Form} \) 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 \( \text{input list} \).
7. The current state \( \text{state} \).
8. An output stream \( \text{output list} \).

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 A.4.1 gives the HOL source code for defining \( TR2 \). Appendix A.4.2 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 33 Execute Command Rule for Transition Relation TR2

\[ \forall \text{inputInterpret certInterpret inputTest certs stateInterpret} \]
\[ x \text{ cmd ins state outStream}. \]
\[ (\forall M \text{ Oi Os).} \]
\[ \text{CFG2Interpret} (M, Oi, Os) \]
\[ (\text{CFG2 inputInterpret certInterpret inputTest certs} \]
\[ \text{stateInterpret} (x::ins) \text{ state outStream}) \Rightarrow \]
\[ (M, Oi, Os) \text{ sat prop (CMD cmd)}) \Rightarrow \]
\[ \forall NS \text{ Out M Oi Os}. \]
\[ \text{TR2} (M, Oi, Os) \text{ (exec (CMD cmd))} \]
\[ (\text{CFG2 inputInterpret certInterpret inputTest certs} \]
\[ \text{stateInterpret} (x::ins) \text{ state outStream}) \]
\[ (\text{CFG2 inputInterpret certInterpret inputTest certs} \]
\[ \text{stateInterpret ins (NS state (exec (CMD cmd))))} \]
\[ (\text{Out state (exec (CMD cmd))::outStream}) \leftrightarrow \]
\[ \text{inputTest} (\text{inputInterpret x}) \land \]
\[ \text{CFG2Interpret} (M, Oi, Os) \]
\[ (\text{CFG2 inputInterpret certInterpret inputTest certs} \]
\[ \text{stateInterpret} (x::ins) \text{ state outStream}) \land \]
\[ (M, Oi, Os) \text{ sat prop (CMD cmd)} \]

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
The three theorems in Figures 32, 33, and 34, 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

\[ \forall M Oi Os. \]
\[ \text{CFG2Interpret } (M, Oi, Os) \]
\[ (\text{CFG2 inputInterpret certInterpret inputTest certs stateInterpret } (x::ins) \text{ state outStream}) \Rightarrow \]
\[ (M, Oi, Os) \text{ sat prop } \text{CMD cmd} \]

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

\[ \forall M Oi Os. \]
\[ \text{CFG2Interpret } (M, Oi, Os) \]
\[ (\text{CFG2 inputInterpret certInterpret inputTest certs stateInterpret } (x::ins) \text{ state outStream}) \Rightarrow \]
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.

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.

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 35 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 \texttt{privcmd} type has three thermostat commands, each of which are security sensitive and require Owner level privileges to execute.

1. \texttt{Set num}, which sets the temperature setting to the number supplied,
2. \texttt{EU}, which enables the \texttt{Utility} to control the thermostat, and
3. \texttt{DU}, which disables the \texttt{Utility} from controlling the thermostat.

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

Figure 35 has seven theorems describing the properties of commands. The \texttt{distinct} theorems state that each command is different than all the others in its type. The \texttt{nchotomy} theorems completely enumerate the values or forms of a member of the particular type can have. The \texttt{11} theorems, e.g., \texttt{set_privcmd_11}, state (where applicable) that identical values have identical components.

### 7.2 Thermostat Principals and Their Privileges

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

<table>
<thead>
<tr>
<th>Principal</th>
<th>Innocuous Commands</th>
<th>Privileged Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Keyboard</td>
<td>Owner</td>
<td>Yes</td>
</tr>
<tr>
<td>Server</td>
<td>Owner</td>
<td>Yes</td>
</tr>
<tr>
<td>Server</td>
<td>Utility</td>
<td>Yes, when Utility is enabled for control. No, otherwise.</td>
</tr>
<tr>
<td>Public keys</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CA</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Owner accounts</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

7.2.1 Principals

Figure 36 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.

7.2.2 Privileges

Principals and their associated privileges are shown in Table 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.
7.3 Thermostat Use Cases

7.3.1 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 37.

When commands come from the Keyboard, the interpretation of what is received is Keyboard | Owner says ⟨command⟩. The thermostat's security context is:

1. The Owner has full authority over all commands, i.e., Owner controls ⟨command⟩.
2. The Keyboard is the Owner's delegate on ⟨command⟩. This is represented as Keyboard reps Owner on ⟨command⟩.

7.3.2 User Control Via the Server

The thermostat is also controlled by the Owner through the Owner's Account on the Server. Figure 38 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 $K_S \mid \text{Owner says } \langle \text{command} \rangle$ by the thermostat.

The thermostat's security context assumes:

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

### 7.3.3 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 $K_U \mid \text{Owner says } \langle \text{priv cmd} \rangle$.

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. $\text{CA controls } (K_U \Rightarrow \text{Utility})$, i.e. the Server trusts CA on public keys.
2. $K_{CA} \Rightarrow \text{Utility})$. This is the public-key certificate for $K_U$ cryptographically signed by $K_{CA}$.
3. $K_{CA} \Rightarrow \text{CA}$. 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 $\langle \text{priv cmd} \rangle$. 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. $\text{Owner controls } (\text{Utility } \mid \text{Owner says } \langle \text{priv cmd} \rangle \supset \text{Utility says } \langle \text{priv cmd} \rangle)$
2. $(\text{Account userid passwd}) \Rightarrow (\text{Utility } \mid \text{Owner says } \langle \text{priv cmd} \rangle \supset \text{Utility says } \langle \text{priv cmd} \rangle)$
3. $(\text{Account userid passwd}) \Rightarrow \text{Owner}$

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 ⟨priv cmd⟩
2. *Server* reps *Owner* on ⟨priv cmd⟩
3. *CA* controls (KS ⇒ *Server*)
4. KCA ⇒ *CA*
5. KCA says (KS ⇒ *Server*)
6. *Server* reps *Utility* on ⟨priv cmd⟩

**Utility Control is Authorized on All Commands** Figure 39 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

\[ \text{Utility controls } \langle \text{NP npriv} \rangle \]

\[ \text{Utility controls } \langle \text{PR privcmd} \rangle \]

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

**Utility Control is Not Authorized on Privileged Commands** Figure 40 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

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

forces privileged commands issued by the Utility to be trapped.

### 7.4 Security Contexts for the Server and Thermostat

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

**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.

### 7.4.2 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. **CA controls** \( K_{CA} \Rightarrow CA \)
6. **CA says** \( K_S \Rightarrow Server \)
7. **Server reps Utility on** \( NP \ npriv \)
8. **Server reps Utility on** \( PR \ privcmd \)
9. **Utility controls** \( NP \ npriv \)

**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 | 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 41 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.
Figure 42 HOL Type Definitions of mode and state

\[
\begin{align*}
\text{mode} & = \text{enabled} \mid \text{disabled} \\
\text{state} & = \text{State mode num}
\end{align*}
\]

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 \(\text{command}\) is instantiated with type \(\text{command}\), as defined in Figure 35.
2. The type variable \(\text{state}\) is instantiated with type \(\text{state}\) defined below.
3. The state interpretation function in \(\text{configuration}\) is instantiated with \(\text{thermoStateInterp}\) defined below.
4. The type variable \(\text{output}\) is instantiated with type \(\text{output}\) defined below.
5. The next-state function \(\text{NS}\) is instantiated with \(\text{thermo1NS}\) defined below.
6. The output function \(\text{Out}\) is instantiated with \(\text{thermo1Out}\) defined below.
7. The certificate list in \(\text{configuration}\) is instantiated with the high-level certificate list \(\text{certs}\), as defined in Figure 41.
8. The authentication function in \(\text{configuration}\) is instantiated with \(\text{isAuthenticated}\) 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, \(\text{discard}\), \(\text{exec}\), and \(\text{trap}\), specialized to the thermostat.

7.5.1 States and Operating Modes

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

We define the thermostat’s \textit{state} as its operating mode and its temperature setting, which we model as a natural number \text{num} in HOL. The type definitions of \text{mode} and \text{state} are in Figure 42.

7.5.2 State Interpretation Function

The interpretation function for thermostat states is given by \(\text{thermoStateInterp}\_\text{def}\) in Figure 43. The definition covers both operating modes:
Thermostat State Interpretation Function

**Figure 43** Thermostat State Interpretation Function

\[ \text{thermoStateInterp utilityID privcmd (State enabled temp)} = \]
\[ \text{Name (Role (Utility utilityID)) controls prop (CMD (PR privcmd))} \]
\[ \land \]
\[ \text{thermoStateInterp utilityID privcmd (State disabled temp)} = \]
\[ \text{Name (Role (Utility utilityID)) says prop (CMD (PR privcmd)) impf prop TRAP} \]

**Figure 44** Transition Diagram for Owner

---

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

\[ \text{thermoStateInterp utilityID privcmd (State enabled temp)} = \]
\[ \text{Name (Role (Utility utilityID)) controls prop (CMD (PR privcmd))} \]

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

\[ \text{thermoStateInterp utilityID privcmd (State disabled temp)} = \]
\[ \text{Name (Role (Utility utilityID)) says prop (CMD (PR privcmd)) impf prop TRAP} \]

The combination of *thermoStateInterp* with the nine access-control logic formulas in *cert*, defined in Figure 41, gives the overall security context for the thermostat’s SSM to authorize authenticated commands.

**7.5.3 Next-State Function**

The next-state transition function for the thermostat can be viewed from the standpoint of the *Owner* and *Utility*. Figures 44 and 45 are the state-transition diagrams for *Owner* and *Utility* originated commands, respectively. Figure 44 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 45 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 46 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 47 shows two theorems nprivSafe and privcmdSecuritySensitive. 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 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.

### 7.5.4 Input Authentication Function

The input authentication function for the top-level thermostat SSM, isAuthenticated, is defined by the HOL source code in Figure 48. 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 ⟨inst⟩, i.e., instructions entered in on the attached keyboard,
Figure 46 Next-State Function for Thermostat

\[ \text{thermoNS} \text{ (State } \text{opMode temp} \text{) discard = State } \text{opMode temp} \] 
\[ \land \] 
\[ \text{exec (CMD (PR (Set newTemp))) = State opMode newTemp} \] 
\[ \land \] 
\[ \text{exec (CMD (PR EU))) = State enabled temp} \] 
\[ \land \] 
\[ \text{exec (CMD (PR DU))) = State disabled temp} \] 
\[ \land \] 
\[ \text{exec (CMD (NP Status))) = State opMode temp} \] 
\[ \land \] 
\[ \text{trap (CMD (PR (Set newTemp))) = State opMode temp} \] 
\[ \land \] 
\[ \text{trap (CMD (PR EU))) = State opMode temp} \] 
\[ \land \] 
\[ \text{trap (CMD (PR DU))) = State opMode temp}\]

Figure 47 Security Properties of Thermostat Commands

\[ \forall \text{npriv state. thermoNS state } \text{(exec (CMD (NP npriv))) = state} \]

\[ \forall \text{privcmd.} \]
\[ \exists \text{state. thermoNS state } \text{(exec (CMD (PR privcmd))) } \neq \text{ state} \]

2. Server \text{| Owner says } \langle \text{inst} \rangle, \text{i.e., the Server relaying instructions from the Owner, and}

3. Server \text{| Utility says } \langle \text{inst} \rangle, \text{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 48. The last clause of the definition containing \text{isAuthenticate} \_ = \text{F} is interpreted by HOL as all other forms as input produce \text{F} as an output. The resulting definition is quite long and appears in Appendix A.4.3.
7.5.5 Output Type and Output Function

Figure 49 Definition of Thermostat Output type and Output Function

\[ \text{output} = \text{report state} \mid \text{flag command} \mid \text{null} \]

\[ \vdash (\text{thermo1Out} (\text{State enabled temp})) = \]
\[ \quad (\text{exec (CMD (PR (Set newTemp)))) = \]
\[ \quad \text{report (State enabled newTemp)} \land \]
\[ (\text{thermo1Out} (\text{State disabled temp})) = \]
\[ \quad (\text{exec (CMD (PR (Set newTemp)))) = \]
\[ \quad \text{report (State disabled newTemp)} \land \]
\[ (\text{thermo1Out} (\text{State enabled temp}) \rightarrow \text{exec (CMD (PR EU))) = \}
\[ \quad \text{report (State enabled temp)) \land \]
\[ (\text{thermo1Out} (\text{State disabled temp}) \rightarrow \text{exec (CMD (PR DU))) = \}
\[ \quad \text{report (State disabled temp)) \land \]
\[ (\text{thermo1Out} (\text{State enabled temp}) \rightarrow \text{exec (CMD (NP Status))) = \}
\[ \quad \text{report (State enabled temp)) \land \]
\[ (\text{thermo1Out} (\text{State disabled temp}) \rightarrow \text{exec (CMD (NP Status))) = \}
\[ \quad \text{report (State disabled temp)) \land \]
\[ (\text{thermo1Out} (\text{State enabled temp}) \rightarrow \text{trap (CMD (PR (Set newTemp))) = \}
\[ \quad \text{flag (PR (Set newTemp))) \land \]
\[ (\text{thermo1Out} (\text{State disabled temp}) \rightarrow \text{trap (CMD (PR (Set newTemp))) = \}
\[ \quad \text{flag (PR (Set newTemp))) \land \]
\[ (\text{thermo1Out} (\text{State enabled temp}) \rightarrow \text{trap (CMD (PR EU))) = \}
\[ \quad \text{flag (PR EU)) \land \]
\[ (\text{thermo1Out} (\text{State disabled temp}) \rightarrow \text{trap (CMD (PR EU))) = \}
\[ \quad \text{flag (PR EU)) \land \]
\[ (\text{thermo1Out} (\text{State enabled temp}) \rightarrow \text{trap (CMD (PR DU))) = \}
\[ \quad \text{flag (PR DU)) \land \]
\[ (\text{thermo1Out} (\text{State disabled temp}) \rightarrow \text{trap (CMD (PR DU))) = \}
\[ \quad \text{flag (PR DU)) \land \]
\[ (\text{thermo1Out} (\text{State enabled temp}) \rightarrow \text{discard = null}) \land \]
\[ (\text{thermo1Out} (\text{State disabled temp}) \rightarrow \text{discard = null}) \land \]
Figure 49 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.

### 7.5.6 Transition Theorems

#### Figure 50 Configuration Interpretation Justifies Executing Keyboarded Command

[CFGInterpret_OWNER_Keyboard_thm]

\[ \forall M \; Oi \; Os . \]
\[ \text{CFGInterpret} \left( M, Oi, Os \right) \]
\[ \left( \text{CFG isAuthenticated} \right) \]
\[ \left( \text{thermoStateInterp utilityID privcmd} \right) \]
\[ \left( \text{certs ownerID utilityID cmd npriv privcmd} \right) \]
\[ \left( \text{Name Keyboard quoting Name (Owner ownerID) says prop (CMD cmd) :: ins} \right) \]
\[ \Rightarrow \left( M, Oi, Os \right) \text{ sat prop (CMD cmd)} \]

#### Figure 51 Executing Keyboarded Commands is Justified

[exec_Keyboard_OWNER_cmd_Justified]

\[ \forall NS \; Out \; outs \; s \; ins \; npriv \; privcmd \; cmd \; ownerID \; utilityID \; M \; Oi \; Os . \]
\[ \text{TR} \left( M, Oi, Os \right) \left( \text{exec (CMD cmd)} \right) \]
\[ \left( \text{CFG isAuthenticated} \right) \]
\[ \left( \text{thermoStateInterp utilityID privcmd} \right) \]
\[ \left( \text{certs ownerID utilityID cmd npriv privcmd} \right) \]
\[ \left( \text{Name Keyboard quoting Name (Owner ownerID) says prop (CMD cmd) :: ins} \right) \]
\[ \left( \text{CFG isAuthenticated} \right) \]
\[ \left( \text{thermoStateInterp utilityID privcmd} \right) \]
\[ \left( \text{certs ownerID utilityID cmd npriv privcmd} \right) \]
\[ \left( \text{NS s (exec (CMD cmd))} \right) \]
\[ \left( \text{Out s (exec (CMD cmd)) :: outs} \right) \Rightarrow \]
\[ \left( M, Oi, Os \right) \text{ sat prop (CMD cmd)} \]
Figure 52 Configuration Interpretation Justifies Executing Owner's Command Via Server

\[ \text{CFGInterpret}_{\text{Owner\_KServer}} \thm \]

\[ \forall M \ Oi \ Os. \\
\text{CFGInterpret} \ (M, Oi, Os) \\
\text{CFG isAuthenticated} \\
\text{(thermoStateInterp utilityID privcmd)} \\
\text{(certs ownerID utilityID cmd npriv privcmd)} \\
\text{(Name (Key (pubK Server)) quoting Name (Owner ownerID) says prop (CMD cmd)::ins) s outs)} \\
\Rightarrow \\
(M, Oi, Os) \ sat \ prop \ (CMD cmd) \]

Figure 53 Executing Owner Command Via Server is Justified

\[ \text{exec\_KServer\_Owner\_cmd\_Justified} \thm \]

\[ \forall NS \ Out \ outs \ s \ ins \ npriv \ privcmd \ cmd \ ownerID \ utilityID \ M \ Oi \ Os. \\
\text{TR} \ (M, Oi, Os) \ (\text{exec (CMD cmd)}) \\
\text{CFG isAuthenticated} \\
\text{(thermoStateInterp utilityID privcmd)} \\
\text{(certs ownerID utilityID cmd npriv privcmd)} \\
\text{(Name (Key (pubK Server)) quoting Name (Owner ownerID) says prop (CMD cmd)::ins) s outs)} \\
\Rightarrow \\
(M, Oi, Os) \ sat \ prop \ (CMD cmd) \]

Figure 54 Configuration Interpretation Justifies Executing Innocuous Utility Command Via Server

\[ \text{CFGInterpret\_Utility\_KServer\_npriv\_thm} \thm \]

\[ \forall M \ Oi \ Os. \\
\text{CFGInterpret} \ (M, Oi, Os) \\
\text{CFG isAuthenticated} \\
\text{(thermoStateInterp utilityID privcmd)} \\
\text{(certs ownerID utilityID cmd npriv privcmd)} \\
\text{(Name (Key (pubK Server)) quoting Name (Role (Utility utilityID)) says prop (CMD (NP npriv))::ins) s outs)} \\
\Rightarrow \\
(M, Oi, Os) \ sat \ prop \ (CMD (NP npriv)) \]
Figure 55 Executing Utility Innocuous Command Via Server is Justified

\[\text{exec\_KServer\_Utility\_npiv\_Justified}\]
\[
\vdash \forall NS \text{ Out outs s ins npiv privcmd cmd ownerID utilityID M Oi Os}.
\]

\[
\begin{align*}
\text{TR} \ (M, Oi, Os) \ (\text{exec} \ (\text{CMD} \ (\text{NP npiv}))) & \\
\{\text{CFG isAuthenticated} & \\
(\text{thermoStateInterp utilityID privcmd}) & \\
(\text{certs ownerID utilityID cmd npiv privcmd}) & \\
(\text{Name (Key (pubK Server)) quoting} & \\
\text{Name (Role (Utility utilityID)) says} & \\
\text{prop} \ (\text{CMD} \ (\text{NP npiv}))::\text{ins} \ s \ outs) & \\
\{\text{CFG isAuthenticated} & \\
(\text{thermoStateInterp utilityID privcmd}) & \\
(\text{certs ownerID utilityID cmd npiv privcmd}) & \\
\text{ins} & \\
(\text{NS s (exec} \ (\text{CMD} \ (\text{NP npiv}))) & \\
(\text{Out s (exec} \ (\text{CMD} \ (\text{NP npiv})))::\text{outs}) & \\
\Rightarrow & \\
\langle M, Oi, Os \rangle \sat \text{prop} \ (\text{CMD} \ (\text{NP npiv})) & \\
\end{align*}
\]

Figure 56 Configuration Interpretation Justifies Executing Utility Privileged Command

\[\text{CFGInterpret\_Utility\_KServer\_privcmd\_thm}\]
\[
\vdash \forall M \ Oi \ Os.
\]

\[
\begin{align*}
\text{CFGInterpret} \ (M, Oi, Os) & \\
\{\text{CFG isAuthenticated} & \\
(\text{thermoStateInterp utilityID privcmd}) & \\
(\text{certs ownerID utilityID cmd npiv privcmd}) & \\
(\text{Name (Key (pubK Server)) quoting} & \\
\text{Name (Role (Utility utilityID)) says} & \\
\text{prop} \ (\text{CMD} \ (\text{PR privcmd}))::\text{ins} & \\
\text{(State enabled temperature} \ \text{outs}) & \\
\Rightarrow & \\
\langle M, Oi, Os \rangle \sat \text{prop} \ (\text{CMD} \ (\text{PR privcmd})) & \\
\end{align*}
\]
Figure 57: Executing Utility Privileged Command Via Server is Justified

\[ \forall \text{NS Out outs temperature ins npriv privcmd cmd ownerID utilityID M Oi Os.} \]

\[
\text{TR (M, Oi, Os) (exec (CMD (PR privcmd)))} \\
\text{(CFG isAuthenticated)} \\
\text{(thermoStateInterp utilityID privcmd)} \\
\text{(certs ownerID utilityID cmd npriv privcmd)} \\
\text{(Name (Key (pubK Server)) quoting Name (Role (Utility utilityID)) says prop (CMD (PR privcmd))::ins)} \\
\text{(State enabled temperature) outs)} \\
\text{(CFG isAuthenticated)} \\
\text{(thermoStateInterp utilityID privcmd)} \\
\text{(certs ownerID utilityID cmd npriv privcmd) ins} \\
\text{(NS (State enabled temperature) exec (CMD (PR privcmd)))} \\
\text{(Out (State enabled temperature) exec (CMD (PR privcmd))::outs))} \Rightarrow \\
\text{(M, Oi, Os) sat prop (CMD (PR privcmd))} \\
\]

Figure 58: Configuration Interpretation Justifies Trapping Utility Privileged Command

\[ \forall \text{M Oi Os.} \]

\[
\text{CFGInterpret (M, Oi, Os)} \\
\text{(CFG isAuthenticated)} \\
\text{(thermoStateInterp utilityID privcmd)} \\
\text{(certs ownerID utilityID cmd npriv privcmd)} \\
\text{(Name (Key (pubK Server)) quoting Name (Role (Utility utilityID)) says prop (CMD (PR privcmd))::ins)} \\
\text{(State disabled temperature) outs) \Rightarrow} \\
\text{(M, Oi, Os) sat prop TRAP} \\
\]
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 50, 52, 54, 56, and 58 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 56 states that the security interpretation of the starting configuration justifies executing the privileged command requested by the Utility via the Server. Specifically, \((M, Oi, Os) \text{ sat prop (CMD (PR privcmd))}\) is derivable from the interpretation \( CFGInterpret \) applied to the configuration shown below.

\[
\begin{align*}
\text{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.}

\text{Figures 50, 52, 54, 56, and 58 show that executing or trapping an instruction is derivable from the security interpretation provided by } CFGInterpret \text{ 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.}

\text{For example, the theorem } CFGInterpret\_Utility\_KServer\_privcmd\_thm \text{ in Figure 56 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) \text{ sat prop (CMD (PR privcmd))} \text{ is derivable from the interpretation } CFGInterpret \text{ applied to the configuration shown below.}

\end{align*}
\]
The theorems in Figures 50, 52, 54, 56, and 58 in conjunction with the $TR_{exec-cmd\_rule}$ theorem, previously proved in Figure 29, give rise to the execution and trap theorems in Figures 51, 53, 55, 57, and 59.

As an example, the $CFGInterpret\_Utility\_KServer\_privcmd\_thm$ theorem, $exec\_KServer\_Utility\_privcmd\_Justified$ shown below and in Figure 57. 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.

\[[exec\_KServer\_Utility\_privcmd\_Justified]\]

\[
\vdash \forall NS Out outs temperature ins npriv privcmd cmd ownerID utilityID M Oi Os.
\begin{align*}
TR (M, Oi, Os) (exec (CMD (PR privcmd))) \\
(CFG isAuthenticated)
\begin{align*}
(thermoStateInterp utilityID privcmd) \\
(certs ownerID utilityID cmd npriv privcmd) \\
(Name (Key (pubK Server)) quoting Name (Role (Utility utilityID)) says prop (CMD (PR privcmd))::ins) \\
(State enabled temperature) outs)
\end{align*}
\begin{align*}
(CFG isAuthenticated)
\begin{align*}
(thermoStateInterp utilityID privcmd) \\
(certs ownerID utilityID cmd npriv privcmd) ins \\
(NS (State enabled temperature)) \\
(exec (CMD (PR privcmd)))
\end{align*}
\begin{align*}
(Out (State enabled temperature)) \\
(exec (CMD (PR privcmd))::outs)) \Rightarrow
\begin{align*}
(M, Oi, Os) sat prop (CMD (PR privcmd))
\end{align*}
\end{align*}
\end{align*}
\]

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 interpretation 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 $cert2Interpret$. 
7.6.1 Thermostat Orders and Messages

**Figure 60 Definitions and Theorems for Orders and Messages**

\[ \text{order} = \text{ORD keyPrinc principal command} \]

\[ [\text{order\_one\_one}] \]
\[ \vdash \forall a_0 a_1 a_2 a_0' a_1' a_2'. \]
\[ (\text{ORD } a_0 a_1 a_2 = \text{ORD } a_0' a_1' a_2') \iff (a_0 = a_0') \land (a_1 = a_1') \land (a_2 = a_2') \]

\[ \text{msg} = \text{KB num command} \]
\[ \mid \text{MSG keyPrinc principal order} \]
\[ ((\text{order digest, keyPrinc) asymMsg}) \]

\[ [\text{msg\_distinct\_thm}] \]
\[ \vdash \forall a_3 a_2 a_1' a_0 a_0. \text{KB } a_0 a_1 \neq \text{MSG } a_0' a_1' a_2 a_3 \]

\[ [\text{msg\_one\_one}] \]
\[ \vdash (\forall a_0 a_1 a_0' a_1'). \]
\[ (\text{KB } a_0 a_1 = \text{KB } a_0' a_1') \iff (a_0 = a_0') \land (a_1 = a_1') \]
\[ \forall a_0 a_1 a_2 a_3 a_0' a_1' a_2' a_3'. \]
\[ (\text{MSG } a_0 a_1 a_2 a_3 = \text{MSG } a_0' a_1' a_2' a_3') \iff (a_0 = a_0') \land (a_1 = a_1') \land (a_2 = a_2') \land (a_3 = a_3') \]

**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 60 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 60. 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.,
where the signature is obtained by signing the hash of the order using the Server's private key, i.e.,

\[ \text{sign} \quad (\text{privK} \ \text{Server}) \quad (\text{hash} \quad (\text{SOME} \quad (\text{ORD} \ \text{Server} \ (\text{Role} (\text{Utility utilityID})) \ (\text{NP Status}))))) \]

The theorems \text{order\_one\_one}, \text{msg\_distinct\_thm}, and \text{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.

### 7.6.2 Authenticating and Checking the Integrity of Messages

**Figure 61 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 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 61 shows the definition of checkmsg
and the theorem \texttt{checkmsg_OK}, where \texttt{checkmsg_OK} shows that \texttt{checkmsg} has the properties we expect.

Looking at the definition of \texttt{checkmsg}, we see three things:

1. \texttt{checkmsg} applied to orders sent over the network via the \texttt{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 \texttt{signVerify}.

2. \texttt{checkmsg} as defined requires the \texttt{sender} value in the message to match the \texttt{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. \texttt{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 manually. Hence, the value of \texttt{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 \texttt{Owners} or their delegates have physical access to a thermostat’s keyboard.

The theorem \texttt{checkmsg_OK} reflects the design decisions on integrity-checking policy and assumptions contained in \texttt{checkmsg}.

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

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

3. Any well-formed keyboard input is regarded as authentic. This reflects the assumption that only the \texttt{Owner} or the \texttt{Owner’s} delegates have physical access to the thermostat’s keyboard.

Section 7.6.3, 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.
7.6.3 Interpreting Messages

**Figure 62 Message Semantics**

\[ \text{msgInterpret}_\text{def} \]

\[ \vdash \text{msgInterpret} \]

\[
\begin{align*}
    & (\text{MSG \, sender \, recipient \, (ORD \, originator \, role \, cmd)} \\
    & \quad \text{signature}) = \\
    & \text{if} \\
    & \quad \text{checkmsg} \\
    & \quad (\text{MSG \, sender \, recipient \, (ORD \, originator \, role \, cmd)} \\
    & \quad \text{signature}) \\
    & \quad \text{then} \\
    & \quad \text{Name \, (Key \, (\text{pubK \, sender})) \, quoting \, Name \, \text{role} \, \text{says}} \\
    & \quad \text{prop \, (CMD \, cmd)} \\
    & \quad \text{else} \\
    & \quad \text{TT}) \land \\
    & \text{(msgInterpret \, (KB \, ownerID \, cmd) =} \\
    & \text{if} \, \text{checkmsg} \, (\text{KB \, ownerID \, cmd}) \, \text{then} \\
    & \quad \text{Name \, Keyboard \, quoting \, Name \, (Owner \, ownerID) \, \text{says}} \\
    & \quad \text{prop \, (CMD \, cmd)} \\
    & \text{else} \, \text{TT})
\end{align*}
\]

**Figure 63 Message Interpretation Theorems**

\[ \text{msgInterpretKB} \]

\[ \vdash (M, Oi, Os) \, \text{sat} \text{msgInterpret} \, (\text{KB \, ownerID \, cmd}) \iff (M, Oi, Os) \, \text{sat} \]

\[
\begin{align*}
    & \text{Name \, Keyboard \, quoting \, Name \, (Owner \, ownerID) \, \text{says}} \\
    & \text{prop \, (CMD \, cmd)}
\end{align*}
\]

\[ \text{msgInterpretMSG_sender_originator_match} \]

\[ \vdash \text{msgInterpret} \]

\[
\begin{align*}
    & (\text{MSG \, sender \, recipient \, (ORD \, sender \, role \, cmd)} \\
    & \quad \text{(sign \, (privK \, sender))} \\
    & \quad \text{(hash \, (SOME \, (ORD \, sender \, role \, cmd)))}) = \\
    & \text{Name \, (Key \, (\text{pubK \, sender})) \, quoting \, Name \, \text{role} \, \text{says}} \\
    & \text{prop \, (CMD \, cmd)}
\end{align*}
\]

\[ \text{msgInterpretMSG_denied} \]

\[ \vdash \text{sender} \neq \text{originator} \Rightarrow (\text{msgInterpret} \]

\[
\begin{align*}
    & (\text{MSG \, sender \, recipient \, (ORD \, originator \, role \, cmd)} \\
    & \quad \text{(sign \, (privK \, sender))} \\
    & \quad \text{(hash \, (SOME \, (ORD \, originator \, role \, cmd)))}) = \\
    & \text{TT}
\end{align*}
\]

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 62 shows the theorem \textit{msgInterpret}_\textit{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

\[
\text{Name (Key (pubK}\,\text{sender))}\,\text{quoting Name}\,\text{role}\,\text{saying}\,\text{prop (CMD}\,\text{cmd})}
\]

If a KB message is authenticated, then its interpretation is

\[
\text{Name Keyboard}\,\text{quoting Name}\,(\text{Owner}\,\text{ownerID})\,\text{saying}\,\text{prop (CMD}\,\text{cmd})}
\]

### 7.6.4 Thermostat Certificates

#### Figure 64 Structure and Integrity-Checking of Thermostat Security Certificates

\[
cert = 
\begin{array}{l}
\text{RCtrCert principal command} \\
| \text{RRepsCert principal principal principal command} \\
| \text{RCtrKCert keyPrinc keyPrinc keyPrinc} \\
| \text{RKeyCert keyPrinc keyPrinc} \\
| \text{KeyCert keyPrinc (keyPrinc pKey) 
(((keyPrinc \times keyPrinc pKey) digest, keyPrinc) asymMsg)}
\end{array}
\]

\[
\vdash (\text{checkcert2}\,(\text{RCtrCert}\,P\,\text{cmd}) \iff T) \land \\
(\text{checkcert2}\,(\text{RRepsCert}\,P\,Q\,\text{cmd}) \iff T) \land \\
(\text{checkcert2}\,(\text{RCtrKCert}\,keyPrinc\,Kq\,keyQpr) \iff T) \land \\
(\text{checkcert2}\,(\text{RKeyCert}\,kp\,keyPpr) \iff T) \land \\
(\text{checkcert2}\,(\text{KeyCert}\,CApr\,Ppr\,(\text{pubK}\,Rpr)\,\text{signature}) \iff \\
\text{signVerify (\text{pubK}\,CApr)\,\text{signature}\,\text{(SOME (Ppr,}\text{pubK}\,Rpr))})
\]
Figure 65 Interpretation of Thermostat Security Certificates

\[\text{cert2Interpret_def} \]

\begin{verbatim}
⊢ (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)
\end{verbatim}
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, $\text{R\text{C}tr\text{C}ert \ P \ cmd}$, interpreted as

   
   $\text{Name } P \ \text{controls prop } (\text{CMD cmd})$

   (b) Delegation certificate, $\text{RReps\text{C}ert \ P \ Q \ cmd}$, 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 64 defines the type cert2 of thermostat certificates described above. The
theorem checkcert2_def in Figure 64 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.

7.6.5 Certificate Interpretation Function

Figure 65 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 66 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 _ _ 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 _ _ cmd) ⇐⇒ T) ∧
   (checkcert2 (RRepsCert _ _ _ cmd) ⇐⇒ T) ∧
   (checkcert2 (RCtrKCert _ _ keyPpr _ keyQpr cmd) ⇐⇒ T) ∧
   (checkcert2 (RKeyCert _ _ keyPpr cmd) ⇐⇒ T) ∧
   (checkcert2 (KeyCert _ _ signature cmd) ⇐⇒
    signVerify (pubK _ _ signature (SOME (Ppr, pubK _ _))))

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

[cert2InterpretRKeyCert]

⊢ (M, Oi, Os) sat cert2Interpret (RKeyCert _ _ cmd) ⇐⇒
   (M, Oi, Os) sat Name (Key (pubK _ _)) speaks_for Name (Role _ _)

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

[cert2InterpretKeyCert]

⊢ (M, Oi, Os) sat
cert2Interpret
   (KeyCert _ _ _ cmd)
   (sign (privK _ _ (hash (SOME (P, pubK _ _)))) cmd) ⇐⇒
   (M, Oi, Os) sat
   Name (Key (pubK _ _)) says
   Name (Key (pubK _ _)) speaks_for Name (Role _ _)

7.6.6 Transition Theorems

**Figure 67 Configuration Interpretation Justifies Executing Keyboarded Command**

\[\forall M Oi Os.\]

\[\text{CFG2Interpret } (M, Oi, Os)\]

\[(\text{CFG2 msgInterpret cert2Interpret isAuthenticated} \]

\[\text{certs2 ownerID utilityID cmd npriv privcmd)}\]

\[(\text{thermoStateInterp utilityID privcmd)}\]

\[(\text{KB ownerID cmd::ins) state outStream)} \Rightarrow\]

\[(M, Oi, Os) \text{ sat prop (CMD cmd)}\]

**Figure 68 Executing Keyboarded Commands is Justified**

\[\forall NS Out M Oi Os.\]

\[\text{TR2 } (M, Oi, Os) \text{ (exec (CMD cmd)}\]

\[(\text{CFG2 msgInterpret cert2Interpret isAuthenticated} \]

\[\text{certs2 ownerID utilityID cmd npriv privcmd)}\]

\[(\text{thermoStateInterp utilityID privcmd)}\]

\[(\text{KB ownerID cmd::ins) state outStream)} \Rightarrow\]

\[(\text{NS state (exec (CMD cmd))}}\]

\[(\text{Out state (exec (CMD cmd))::outStream) } \Rightarrow\]

\[(M, Oi, Os) \text{ sat prop (CMD cmd)}\]

**Figure 69 Configuration Interpretation Justifies Executing Owner’s Command Via Server**

\[\forall M Oi Os.\]

\[\text{CFG2Interpret } (M, Oi, Os)\]

\[(\text{CFG2 msgInterpret cert2Interpret isAuthenticated} \]

\[\text{certs2 ownerID utilityID cmd npriv privcmd)}\]

\[(\text{thermoStateInterp utilityID privcmd)}\]

\[(\text{MSG Server (Owner ownerID)}}\]

\[(\text{ORD Server (Owner ownerID) cmd)}\]

\[(\text{sign (privK Server)}\]

\[(\text{hash} \]

\[(\text{SOME (ORD Server (Owner ownerID) cmd))::ins) state outStream) } \Rightarrow\]

\[(M, Oi, Os) \text{ sat prop (CMD cmd)}\]
Figure 70 Executing Owner Command Via Server is Justified

\[ \text{[exec2\_KServer\_Owner\_cmd\_Justified]} \]

\[ \vdash \forall M \text{ Out } Oi \text{ Os}. \]

\[ \text{TR2} \ (M, Oi, Os) \ (\text{exec} (\text{CMD } cmd)) \]

\[ (\text{CFG2 msgInterpret cert2Interpret isAuthenticated}) \]

\[ (\text{certs2 } ownerID \text{ utilityID cmd npriv privcmd}) \]

\[ (\text{thermoStateInterp utilityID privcmd}) \]

\[ (\text{MSG Server (Owner } ownerID)) \]

\[ (\text{ORD Server (Owner } ownerID) \text{ cmd}) \]

\[ (\text{sign (privK Server}) \]

\[ (\text{hash}) \]

\[ (\text{SOME (ORD Server (Owner } ownerID) \text{ cmd})} ):: \]

\[ \text{ins) state outStream}) \]

\[ (\text{CFG2 msgInterpret cert2Interpret isAuthenticated}) \]

\[ (\text{certs2 } ownerID \text{ utilityID cmd npriv privcmd}) \]

\[ (\text{thermoStateInterp utilityID privcmd}) \text{ ins} \]

\[ (\text{NS state} \ (\text{exec} (\text{CMD } cmd)))) \]

\[ (\text{Out state} \ (\text{exec} \ (\text{CMD } cmd}) ):: \text{outStream}) \Rightarrow \]

\[ (M, Oi, Os) \text{ sat prop } (\text{CMD } cmd) \]

---

Figure 71 Configuration Interpretation Justifies Executing Innocuous Utility Command Via Server

\[ \text{[CFG2\_Interpret\_Utility\_KServer\_npriv\_thm]} \]

\[ \vdash \forall M \text{ Oi Os}. \]

\[ \text{CFG2Interpret} \ (M, Oi, Os) \]

\[ (\text{CFG2 msgInterpret cert2Interpret isAuthenticated}) \]

\[ (\text{certs2 } ownerID \text{ utilityID cmd npriv privcmd}) \]

\[ (\text{thermoStateInterp utilityID privcmd}) \]

\[ (\text{MSG Server (Role (Utility } utilityID))) \]

\[ (\text{ORD Server (Role (Utility } utilityID)) \ (\text{NP npriv})) \]

\[ (\text{sign (privK Server}) \]

\[ (\text{hash}) \]

\[ (\text{SOME} \]

\[ (\text{ORD Server (Role (Utility } utilityID)) \]

\[ (\text{NP npriv}))} ):: \text{ins}) \text{ state outStream}) \Rightarrow \]

\[ (M, Oi, Os) \text{ sat prop } (\text{CMD (NP npriv)}) \]
**Figure 72** Executing Utility Innocuous Command Via Server is Justified

\[ \exists \text{exec}_2_{KServer\_Utility\_npriv\_Justified} \]

\[ \forall \text{NS} \text{Out} \text{outStream} \text{state} \text{ins} \text{npriv} \text{privcmd} \text{cmd} \text{ownerID} \text{utilityID} \text{M} \text{Oi} \text{Os}. \]

\[ \text{TR2} (M, Oi, Os) \ (\text{exec} \ (\text{CMD} \ (\text{NP} \ npriv))) \]

\[ (\text{CFG2 msgInterpret cert2Interpret isAuthenticated} \]

\[ (\text{certs2 ownerID utilityID cmd npriv privcmd}) \]

\[ (\text{thermoStateInterp utilityID privcmd}) \]

\[ (\text{MSG Server (Role (Utility utilityID))}) \]

\[ (\text{ORD Server (Role (Utility utilityID))}) \ (\text{NP npriv})) \]

\[ \text{(sign (privK Server))} \]

\[ \text{(hash} \]

\[ (\text{SOME} \]

\[ (\text{ORD Server (Role (Utility utilityID))}) \ (\text{NP npriv}))::ins) \text{ state outStream}) \]

\[ (\text{CFG2 msgInterpret cert2Interpret isAuthenticated} \]

\[ (\text{certs2 ownerID utilityID cmd npriv privcmd}) \]

\[ (\text{thermoStateInterp utilityID privcmd}) \]

\[ (\text{NS state (exec (CMD (NP npriv)))}) \]

\[ (\text{Out state (exec (CMD (NP npriv)))::outStream}) \Rightarrow \]

\[ (M, Oi, Os) \text{ sat prop (CMD (NP npriv))} \]

---

**Figure 73** Configuration Interpretation Justifies Executing Utility Privileged Command

\[ \exists \text{CFG2Interpret\_Utility\_KServer\_privcmd\_thm} \]

\[ \forall \text{M} \text{Oi} \text{Os}. \]

\[ \text{CFG2Interpret} (M, Oi, Os) \]

\[ (\text{CFG2 msgInterpret cert2Interpret isAuthenticated} \]

\[ (\text{certs2 ownerID utilityID cmd npriv privcmd}) \]

\[ (\text{thermoStateInterp utilityID privcmd}) \]

\[ (\text{MSG Server (Role (Utility utilityID))}) \]

\[ (\text{ORD Server (Role (Utility utilityID))}) \ (\text{PR privcmd}) \]

\[ \text{(sign (privK Server))} \]

\[ \text{(hash} \]

\[ (\text{SOME} \]

\[ (\text{ORD Server (Role (Utility utilityID))}) \ (\text{PR privcmd})})::ins) \]

\[ \text{(State enabled temperature) outStream}) \Rightarrow \]

\[ (M, Oi, Os) \text{ sat prop (CMD (PR privcmd))} \]
Figure 74 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\}\}\}\}\}\}
\{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\}\}\}\}\}\)
⇒ (M, Oi, Os) sat prop (CMD (PR privcmd))

Figure 75 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\}\}\}\}\}\}
\{State disabled temperature\} outStream) ⇒
(M, Oi, Os) sat prop TRAP
Figure 76 Trapping Utility Privileged Command Via Server is Justified

\[ \forall NS \text{ Out outStream temperature ins npriv privcmd cmd ownerID utilityID } M Oi Os . \]

\[ \text{TR2 } (M, Oi, Os) (\text{trap } (\text{CMD } (\text{PR } \text{privcmd}))) \]

\[
(M, Oi, Os) \Rightarrow \text{sat prop TRAP}
\]

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 67, 69, 71, 73, and 75. 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.

\[ \forall M Oi Os . \]

\[
\text{CFG2Interpret } (M, Oi, Os) \]

\[
(M, Oi, Os) \Rightarrow \text{sat prop TRAP}
\]
The refined execution theorems corresponding to the top-level SSM execution theorems are in Figures 68, 70, 72, 74, and 76. As an example, the \texttt{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 \textit{Utility}, then the execution was justified.

\[
\texttt{exec2\_KServer\_Utility\_privcmd\_Justified}
\]

\[
\forall \text{NS Out outStream temperature ins npriv privcmd cmd ownerID utilityID M Oi Os}.
\]

\[
\text{TR2 (M, Oi, Os) (exec (CMD (PR privcmd)))}
\]

\[
(CFG2 \text{msgInterpret cert2Interpret isAuthenticated}
\]

\[
(certs2 ownerID utilityID cmd npriv privcmd)
\]

\[
(thermoStateInterp utilityID privcmd)
\]

\[
(MSG Server (Role (Utility utilityID))
\]

\[
(ORD Server (Role (Utility utilityID))
\]

\[
(PR privcmd)
\]

\[
(sign (privK Server)
\]

\[
(hash
\]

\[
(SOME
\]

\[
(ORD Server (Role (Utility utilityID))
\]

\[
(PR privcmd)))))::\texttt{ins}
\]

\[
(\text{State enabled temperature) outStream}) \Rightarrow
\]

\[
(M, Oi, Os) \text{ sat prop (CMD (PR privcmd))}
\]
7.7 Equivalence of Top-Level and Refined Secure State-Machines
Figure 78 TR and TR2 Equivalence for Owner Commands Via Server

\[
\vdash \forall M, O_i, O_s \ \text{ownerID utilityID ins ins}_2 \ \text{outStream NS Out state npriv privcmd cmd}.
\]

\[
\text{TR2 } (M, O_i, O_s) \ (\text{exec } (\text{CMD cmd}))
\]

\[
(\text{CFG2 msgInterpret cert2Interpret isAuthenticated}
\]
\[
(\text{certs2 ownerID utilityID cmd npriv privcmd})
\]
\[
(\text{thermoStateInterp utilityID privcmd})
\]
\[
(\text{MSG Server (Owner ownerID)})
\]
\[
(\text{ORD Server (Owner ownerID) cmd})
\]
\[
(\text{sign (privK Server})
\]
\[
(\text{hash}
\]
\[
(\text{SOME (ORD Server (Owner ownerID) cmd)}))::
\]
\[
\text{ins}_2 \ \text{state outStream})
\]

\[
(\text{CFG2 msgInterpret cert2Interpret isAuthenticated}
\]
\[
(\text{certs2 ownerID utilityID cmd npriv privcmd})
\]
\[
(\text{thermoStateInterp utilityID privcmd}) \ \text{ins}_2
\]
\[
(\text{NS state } (\text{exec } (\text{CMD cmd}))
\]
\[
(\text{Out state } (\text{exec } (\text{CMD cmd}))::\text{outStream})) \iff
\]

\[
\text{TR } (M, O_i, O_s) \ (\text{exec } (\text{CMD cmd}))
\]

\[
(\text{CFG isAuthenticated}
\]
\[
(\text{thermoStateInterp utilityID privcmd})
\]
\[
(\text{certs ownerID utilityID cmd npriv privcmd})
\]
\[
(\text{Name (Key (pubK Server)) quoting}
\]
\[
\text{Name (Owner ownerID) says prop (CMD cmd)::ins) state outStream})
\]

\[
(\text{CFG isAuthenticated}
\]
\[
(\text{thermoStateInterp utilityID privcmd})
\]
\[
(\text{certs ownerID utilityID cmd npriv privcmd}) \ \text{ins}
\]
\[
(\text{NS state } (\text{exec } (\text{CMD cmd}))
\]
\[
(\text{Out state } (\text{exec } (\text{CMD cmd}))::\text{outStream}))
\]
Figure 79 TR and TR2 Equivalence for Utility Non-Privileged Commands Via Server

\[ \text{[TR2 iff TR\_KServer\_Utility\_npriv]} \]

\[ \vdash \forall M \, O_i \, O_s \, \text{ownerID utilityID} \, \text{ins} \, \text{ins}_2 \, \text{outStream} \, \text{NS} \, \text{Out state npriv privcmd cmd}. \]

\[ \text{TR2} \, (M, O_i, O_s) \, (\text{exec} \, \text{CMD} \, (\text{NP npriv})) \]

\[ \text{(CFG2 msgInterpret cert2Interpret isAuthenticated} \]

\[ \text{(certs2 ownerID utilityID cmd npriv privcmd)} \]

\[ \text{(thermoStateInterp utilityID privcmd)} \]

\[ \text{(MSG Server (Role (Utility utilityID))} \}

\[ \text{(ORD Server (Role (Utility utilityID))) \, (NP npriv))} \]

\[ \text{(sign (privK Server)} \]

\[ \text{(hash}\]

\[ \text{(SOME} \]

\[ \text{(ORD Server (Role (Utility utilityID))} \]

\[ \text{(NP npriv))}}::\text{ins}_2 \, \text{state outStream)} \]

\[ \text{(CFG msgInterpret cert2Interpret isAuthenticated} \]

\[ \text{(certs2 ownerID utilityID cmd npriv privcmd)} \]

\[ \text{(thermoStateInterp utilityID privcmd)} \, \text{ins}_2 \]

\[ \text{(NS state} \, \text{exec} \, \text{CMD} \, (\text{NP npriv}))} \]

\[ \text{(Out state} \, \text{exec} \, \text{CMD} \, (\text{NP npriv))}}::\text{outStream)} \]

\[ \iff \]

\[ \text{TR} \, (M, O_i, O_s) \, \text{(exec} \, \text{CMD} \, (\text{NP npriv}))} \]

\[ \text{(CFG isAuthenticated} \]

\[ \text{(thermoStateInterp utilityID privcmd)} \]

\[ \text{(certs ownerID utilityID cmd npriv privcmd)} \]

\[ \text{(Name (Key (pubK Server)) quoting} \]

\[ \text{Name (Role (Utility utilityID)) says} \]

\[ \text{prop} \, \text{CMD} \, (\text{NP npriv))}}::\text{ins} \, \text{state outStream)} \]

\[ \text{(CFG isAuthenticated} \]

\[ \text{(thermoStateInterp utilityID privcmd)} \]

\[ \text{(certs ownerID utilityID cmd npriv privcmd) \, \text{ins}} \]

\[ \text{(NS state} \, \text{exec} \, \text{CMD} \, (\text{NP npriv}))} \]

\[ \text{(Out state} \, \text{exec} \, \text{CMD} \, (\text{NP npriv))}}::\text{outStream)} \]
Figure 80 TR and TR2 Equivalence for Utility Privileged Commands Via Server

\[ \text{TR2 iff TR2}_{\text{KServer Utility privcmd}} \]

\( \vdash \forall M \, Oi \, Os \, \text{ownerID utilityID ins ins}_2 \, \text{temperature outStream NS} \)

\[ \text{Out npnpriv privcmd cmd}. \]

\( \text{TR2} \ (M, Oi, Os) \ (\text{exec} \ (\text{CMD} \ (\text{PR} \ \text{privcmd}))) \)

\[ \text{(CFG} \ \text{msgInterpret} \ \text{cert2Interpret} \ \text{isAuthenticated} \]

\[ \text{(certs}2 \ \text{ownerID utilityID cmd npnpriv privcmd)} \]

\[ \text{(thermoStateInterp utilityID privcmd)} \]

\[ \text{(MSG Server (Role (Utility utilityID))} \]

\[ \text{(ORD Server (Role (Utility utilityID))} \]

\[ \text{(PR privcmd)} \]

\[ \text{(sign (privK Server)} \]

\[ \text{(hash} \]

\[ \text{(SOME} \]

\[ \text{(ORD Server (Role (Utility utilityID))} \]

\[ \text{(PR privcmd)})::ins}_2 \]

\[ \text{(State enabled temperature) outStream} \]

\( \text{CFG} \ \text{msgInterpret} \ \text{cert2Interpret} \ \text{isAuthenticated} \]

\[ \text{(certs}2 \ \text{ownerID utilityID cmd npnpriv privcmd)} \]

\[ \text{(thermoStateInterp utilityID privcmd) ins}_2 \]

\[ \text{(NS (State enabled temperature)} \]

\[ \text{(exec} \ (\text{CMD} \ (\text{PR} \ \text{privcmd}))) \]

\[ \text{(Out (State enabled temperature)} \]

\[ \text{(exec} \ (\text{CMD} \ (\text{PR} \ \text{privcmd})))::\text{outStream}) \ \iff \]

\[ \text{TR} \ (M, Oi, Os) \ (\text{exec} \ (\text{CMD} \ (\text{PR} \ \text{privcmd}))) \]

\[ \text{(CFG isAuthenticated} \]

\[ \text{(thermoStateInterp utilityID privcmd)} \]

\[ \text{(certs ownerID utilityID cmd npnpriv privcmd)} \]

\[ \text{(Name (Key (pubK Server)) quoting} \]

\[ \text{Name (Role (Utility utilityID)) says} \]

\[ \text{prop (CMD (PR privcmd))::ins} \]

\[ \text{(State enabled temperature) outStream} \]

\[ \text{CFG isAuthenticated} \]

\[ \text{(thermoStateInterp utilityID privcmd)} \]

\[ \text{(certs ownerID utilityID cmd npnpriv privcmd) ins} \]

\[ \text{(NS (State enabled temperature)} \]

\[ \text{(exec} \ (\text{CMD} \ (\text{PR} \ \text{privcmd}))) \]

\[ \text{(Out (State enabled temperature)} \]

\[ \text{(exec} \ (\text{CMD} \ (\text{PR} \ \text{privcmd})))::\text{outStream}) \]
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 77, 78, 79, 80, and 81 have the five theorems.
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.
A Appendix

A.1 The Definition of ACL Formulas, Kripke Structures, Principals, Integrity Levels, and Security Levels in HOL

Form =
TT
| FF
| prop 'aavar
| notf (('aavar, 'apn, 'il, 'sl) Form)
| (andf) (('aavar, 'apn, 'il, 'sl) Form)
| (orf) (('aavar, 'apn, 'il, 'sl) Form)
| (impf) (('aavar, 'apn, 'il, 'sl) Form)
| (eqf) (('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)
| (domi) ('apn, 'il) IntLevel)
| (eqi) ('apn, 'il) IntLevel)
| (eqs) ('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
A.2 The Definition of the Evaluation Function $\mathcal{E}_M$ in HOL

The semantics or values of well-formed access-control logic formulas in HOL, is defined by $E$. 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 $M$.

[Ev_def]

\[ \vdash (\forall Oi Os M . \ Efn Oi Os M \ TT = \mathcal{U}(:'v)) \land \\
(\forall Oi Os M p . \ Efn Oi Os M (\text{prop } p) = \text{intpKS } M p) \land \\
(\forall Oi Os M f . \ Efn Oi Os M (\text{notf } f) = \mathcal{U}(:'v) \ \text{DIFF} \ Efn Oi Os M f) \land \\
(\forall Oi Os M f_1 f_2 . \ Efn Oi Os M (f_1 \ \text{andf } f_2) = \\
Efn Oi Os M f_1 \cap Efn Oi Os M f_2) \land \\
(\forall Oi Os M f_1 f_2 . \ Efn Oi Os M (f_1 \ \text{orf } f_2) = \\
Efn Oi Os M f_1 \cup Efn Oi Os M f_2) \land \\
(\forall Oi Os M f_1 f_2 . \ Efn Oi Os M (f_1 \ \text{impf } f_2) = \\
\mathcal{U}(:'v) \ \text{DIFF} \ Efn Oi Os M f_1 \cup Efn Oi Os M f_2) \land \\
(\forall Oi Os M f . \ Efn Oi Os M (P \ \text{says } f) = \\
\{w | Jext (jKS M) P w \subseteq Efn Oi Os M f\}) \land \\
(\forall Oi Os M P Q . \ Efn Oi Os M (P \ \text{speaks_for } Q) = \\
\text{if } Jext (jKS M) Q \ R\text{SUBSET} Jext (jKS M) P \ \text{then } \mathcal{U}(:'v) \ \text{else } \{\}) \land \\
(\forall Oi Os M P f . \ Efn Oi Os M (P \ \text{controls } f) = \\
\mathcal{U}(:'v) \ \text{DIFF} \ \{w | Jext (jKS M) P w \subseteq Efn Oi Os M f\} \cup \\
Efn Oi Os M f) \land \\
(\forall Oi Os M P Q f . \ Efn Oi Os M (reps } P Q f) = \\
\mathcal{U}(:'v) \ \text{DIFF} \ \\
\{w | Jext (jKS M) (P \ \text{quoting } Q) w \subseteq Efn Oi Os M f\} \cup \\
\{w | Jext (jKS M) Q w \subseteq Efn Oi Os M f\}) \land \\
(\forall Oi Os M intl_1 intl_2 . \ Efn Oi Os M (intl_1 \ \text{domi } intl_2) = \\
\text{if } \text{repPO } Oi (\text{Lifn } M intl_2) (\text{Lifn } M intl_1) \ \text{then } \mathcal{U}(:'v) \ \text{else } \{\}) \land \\
(\forall Oi Os M intl_2 intl_1 . \ Efn Oi Os M (intl_2 \ \text{eqi } intl_1) = \\
\{\text{if } \text{repPO } Oi (\text{Lifn } M intl_2) (\text{Lifn } M intl_1) \ \text{then } \mathcal{U}(:'v) \ \text{else } \{\}) \land \\
} \]
A.3 Definition of Transition Relation TR

A.3.1 HOL Source Code Defining TR

```haskell
val [TR_rules, TR_cases] = HOL_rules
  (inputTest (P says (prop (CMD cmd))): (command inst, principal, d, e):Form) = if repPO Os (Lsfn M secl2) then \ U(\'v) else \ }
  (\forall Os M secl2. Efn Os M (secl2 DOM secl2) = if repPO Os (Lsfn M secl1) then \ U(\'v) else \ }
  (\forall Os M secl2 secl1. Efn Os M M (secl2 eqn secl1) = if repPO Os (Lsfn M secl2) then \ U(\'v) else \ }
  (\forall Os M numExp1 numExp2. Efn Os M M (numExp1 eqn numExp2) = if numExp1 = numExp2 then \ U(\'v) else \ }
  (\forall Os M numExp1 numExp2. Efn Os M M (numExp1 lte numExp2) = if numExp1 \\ numExp2 then \ U(\'v) else \ }
  (\forall Os M numExp1 numExp2. Efn Os M M (numExp1 lt numExp2) = if numExp1 < numExp2 then \ U(\'v) else \ }

\begin{align*}
\text{if repPO } Oi \ (\text{Lifn } M \text{ int}_{12}) \ (\text{Lifn } M \text{ int}_{12}) \text{ then } & U(\'v) \\
\text{else } & \}
\end{align*}
```

```haskell
\begin{align*}
(\forall Oi Os M secl1 secl2. Efn Oi Os M (secl1 DOM secl2) = \\
\text{if repPO } Oi \ (\text{Lsfn } M \text{ secl2}) \ (\text{Lsfn } M \text{ secl1}) \text{ then } & U(\'v) \\
\text{else } & \}
\end{align*}
```

```haskell
\begin{align*}
(\forall Oi Os M secl2 secl1. Efn Oi Os M (secl2 eqn secl1) = \\
\text{if repPO } Oi \ (\text{Lsfn } M \text{ secl2}) \ (\text{Lsfn } M \text{ secl1}) \text{ then } & U(\'v) \\
\text{else } & \}
\end{align*}
```

```haskell
\begin{align*}
(\forall Oi Os M numExp1 numExp2. Efn Oi Os M M (numExp1 eqn numExp2) = \\
\text{if numExp1 = numExp2 then } & U(\'v) \\
\text{else } & \}
\end{align*}
```

```haskell
\begin{align*}
(\forall Oi Os M numExp1 numExp2. Efn Oi Os M M (numExp1 lte numExp2) = \\
\text{if numExp1 \\ numExp2 then } & U(\'v) \\
\text{else } & \}
\end{align*}
```

```haskell
\begin{align*}
(\forall Oi Os M numExp1 numExp2. Efn Oi Os M M (numExp1 lt numExp2) = \\
\text{if numExp1 \\ numExp2 then } & U(\'v) \\
\text{else } & \}
\end{align*}
```
A.3.2 Defining Properties of TR

[TR_rules]
\[\vdash (\forall \text{inputTest } P \text{ NS } M \text{ Oi } Os \text{ Out } s \text{ certs } \text{stateInterp } \text{cmd } \text{ins } \text{outs}).\]
\[\text{inputTest } (P \text{ says prop } (\text{CMD cmd})) \land\]
\[\text{CFGInterpret } (M, Oi, Os)\]
\[\text{CFG inputTest stateInterp certs}\]
\[\text{(P says prop } (\text{CMD cmd})::\text{ins} \text{ s outs}) \Rightarrow\]
\[\text{TR } (M, Oi, Os) (\text{exec } (\text{CMD cmd}))\]
\[\text{CFG inputTest stateInterp certs}\]
\[\text{(P says prop } (\text{CMD cmd})::\text{ins} \text{ s outs})\]
\[\text{CFG inputTest stateInterp certs ins}\]
\[\text{(NS s } (\text{exec } (\text{CMD cmd})))\]
\[\text{(Out s } (\text{exec } (\text{CMD cmd})):::\text{outs}))) \land\]
\[\forall \text{inputTest } P \text{ NS } M \text{ Oi } Os \text{ Out } s \text{ certs } \text{stateInterp } \text{cmd } \text{ins } \text{outs}.\]
\[\text{inputTest } (P \text{ says prop } (\text{CMD cmd})) \land\]
\[\text{CFGInterpret } (M, Oi, Os)\]
\[\text{CFG inputTest stateInterp certs}\]
\[\text{(P says prop } (\text{CMD cmd})::\text{ins} \text{ s outs}) \Rightarrow\]
\[\text{TR } (M, Oi, Os) (\text{trap } (\text{CMD cmd}))\]
\[\text{CFG inputTest stateInterp certs}\]
\[\text{(P says prop } (\text{CMD cmd})::\text{ins} \text{ s outs})\]
\[\text{CFG inputTest stateInterp certs ins}\]
\[\text{(NS s } (\text{trap } (\text{CMD cmd})))\]
\[\text{(Out s } (\text{trap } (\text{CMD cmd})):::\text{outs}))) \land\]
\[\forall \text{inputTest } NS \text{ M } Oi \text{ Os } Out \text{ s } \text{certs stateInterp } \text{cmd } \text{x } \text{ins outs}.\]
\[\neg \text{inputTest } x \Rightarrow\]
\[\text{TR } (M, Oi, Os) \text{ discard}\]
\[\text{CFG inputTest stateInterp certs } (x::\text{ins} \text{ s outs})\]
\[\text{CFG inputTest stateInterp certs ins } (\text{NS s discard})\]
\[\text{(Out s discard::outs})\]

[TR_ind]
\[\vdash \forall \text{TR'} .\]
\[\forall \text{inputTest } P \text{ NS } M \text{ Oi } Os \text{ Out } s \text{ certs stateInterp } \text{cmd } \text{ins } \text{outs}.\]
\[\text{inputTest } (P \text{ says prop } (\text{CMD cmd})) \land\]
\[\text{CFGInterpret } (M, Oi, Os)\]
\[\text{CFG inputTest stateInterp certs}\]
\[\text{(P says prop } (\text{CMD cmd})::\text{ins} \text{ s outs}) \Rightarrow\]
\[\text{TR'} (M, Oi, Os) (\text{exec } (\text{CMD cmd}))\]
\[\text{CFG inputTest stateInterp certs}\]
\[\text{(P says prop } (\text{CMD cmd})::\text{ins} \text{ s outs})\]
\begin{align*}
&(\text{CFG inputTest stateInterp certs ins}) \\
&(\text{NS s (exec (CMD cmd))}) \\
&(\text{Out s (exec (CMD cmd)::outs))}) \\
&(\text{\forall \text{inputTest P NS M Oi Os Out s certs stateInterp cmd ins outs}}) \\
&\text{inputTest (P says prop (CMD cmd))} \\
&\text{CFGInterpret (M, Oi, Os)} \\
&(\text{CFG inputTest stateInterp certs}) \\
&(\text{P says prop (CMD cmd)::ins s outs}) \Rightarrow \\
&\text{TR'} (M, Oi, Os) (\text{trap (CMD cmd)}) \\
&(\text{CFG inputTest stateInterp certs}) \\
&(\text{P says prop (CMD cmd)::ins s outs}) \\
&(\text{CFG inputTest stateInterp certs ins}) \\
&(\text{NS s (trap (CMD cmd))}) \\
&(\text{Out s (trap (CMD cmd)::outs))}) \\
&(\text{\forall \text{inputTest P NS M Oi Os Out s certs stateInterp cmd x ins outs}}) \\
&\sim\text{inputTest x } \Rightarrow \\
&\text{TR'} (M, Oi, Os) \text{ discard} \\
&(\text{CFG inputTest stateInterp certs (x::ins s outs}) \\
&(\text{CFG inputTest stateInterp certs ins (NS s discard}) \\
&(\text{Out s discard::outs))}) \Rightarrow \\
&\forall a_0 a_1 a_2 a_3. \text{ TR } a_0 a_1 a_2 a_3 \Rightarrow TR' a_0 a_1 a_2 a_3
\end{align*}

[TR_cases]
\begin{align*}
\vdash &\forall a_0 a_1 a_2 a_3. \\
&\text{TR } a_0 a_1 a_2 a_3 \iff \\
&(\exists \text{inputTest P NS M Oi Os Out s certs stateInterp cmd ins outs}) \\
&(a_0 = (M, Oi, Os)) \land (a_1 = \text{exec } (\text{CMD cmd})) \land \\
&(a_2 = \text{CFG inputTest stateInterp certs}) \\
&(P \text{ says prop } (\text{CMD cmd)::ins s outs}) \land \\
&(a_3 = \text{CFG inputTest stateInterp certs ins}) \\
&(\text{NS s } (\text{exec } (\text{CMD cmd}))) \\
&(\text{Out s } (\text{exec } (\text{CMD cmd})::\text{outs})) \land \\
&\text{inputTest (P says prop } (\text{CMD cmd})) \land \\
&\text{CFGInterpret (M, Oi, Os)} \\
&(\text{CFG inputTest stateInterp certs}) \\
&(\text{P says prop } (\text{CMD cmd)::ins s outs}) \lor \\
&(\exists \text{inputTest P NS M Oi Os Out s certs stateInterp cmd ins outs}) \\
&(a_0 = (M, Oi, Os)) \land (a_1 = \text{trap } (\text{CMD cmd})) \land \\
&(a_2 = \text{CFG inputTest stateInterp certs}) \\
&(P \text{ says prop } (\text{CMD cmd)::ins s outs}) \land \\
&(a_3 = \text{CFG inputTest stateInterp certs ins}) \\
&(\text{NS s } (\text{trap } (\text{CMD cmd})))
\end{align*}
\[(\text{Out } s \ (\text{Trap } \text{CMD cmd}) :: \text{outs}) \land \\ \text{inputTest } (P \text{ says prop } (\text{CMD cmd})) \land \\ \text{CFGInterpret } (M, O_i, O_s) \\ \text{CFG inputTest stateInterp certs } \\ (P \text{ says prop } (\text{CMD cmd}) :: \text{ins} \ s \ \text{outs}) \lor \\ \exists \text{inputTest } NS \ M \ O_i \ O_s \ Out \ s \ \text{certs} \ \text{stateInterp} \ \text{cmd} \ x \ \text{ins} \ \text{outs}. \]

\[(a_0 = (M, O_i, O_s)) \land (a_1 = \text{discard}) \land \\ (a_2 = \text{CFG inputTest stateInterp certs} (x :: \text{ins}) \ s \ \text{outs}) \land \\ (a_3 = \\ \text{CFG inputTest stateInterp certs} \ \text{ins} \ (NS \ s \ \text{discard}) \\ (\text{Out } s \ \text{discard} :: \text{outs}) \land \lnot \text{inputTest } x) \]

### A.4 Definition of Transition Relation TR2

#### A.4.1 HOL Source Code Defining TR2

```haskell
val (TR2_rules, TR2_ind, TR2_cases) = 
HOL.rel "(\forall (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) 
(stateInterpret : 'state -> ('command inst, 'principal, 'd, 'e)Form) 
outStream : 'output list) 
\[\text{inputTest}(\text{inputInterpret } (x :: 'input)) \land \]
\[(\text{CFGInterpret} (M, O_i, O_s) \text{ inputTest } \text{certs} \ \text{stateInterp} \ \text{ins} \ (NS \ state \ \text{outStream}) \Rightarrow 
\text{TR2} (M, O_i, O_s) \ \text{(execute(CMD cmd))}) \]
\[(\text{CFGInterpret} (M, O_i, O_s) \text{ inputTest } \text{certs} \ \text{stateInterp} \ \text{ins} \ (NS \ state \ \text{outStream})) \Rightarrow 
\text{TR2} (M, O_i, O_s) \ \text{(execute(CMD cmd))}) \]
\text{inputTest}\text{inputInterpret}(x :: 'input) \land 
\[(\text{CFGInterpret} (M, O_i, O_s) \text{ inputTest } \text{certs} \ \text{stateInterp} \ \text{ins} \ (NS \ state \ \text{outStream})) \Rightarrow 
\text{TR2} (M, O_i, O_s) \ \text{(execute(CMD cmd))}) \]
```

A.4.2 Defining Properties of TR2

[TR2_rules]

\[\vdash (\forall \text{inputInterpret certInterpret inputTest x NS M Oi Os Out state certs stateInterpret cmd ins outStream}.\]

inputTest (inputInterpret x) \land

CFG2Interpret (M, Oi, Os)

(CFG2 inputInterpret certInterpret inputTest certs stateInterpret (x::ins) state outStream) \Rightarrow

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)) \land

(\forall \text{inputInterpret certInterpret inputTest x NS M Oi Os Out state certs stateInterpret cmd ins outStream}.)

inputTest (inputInterpret x) \land

CFG2Interpret (M, Oi, Os)

(CFG2 inputInterpret certInterpret inputTest certs stateInterpret (x::ins) state outStream) \Rightarrow

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)) \land

\forall \text{inputInterpret certInterpret inputTest x NS M Oi Os Out state certs stateInterpret cmd ins outStream}.\]

\neg inputTest (inputInterpret x) \Rightarrow

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))
\[
\frac{\forall \text{inputInterpret certInterpret inputTest x NS M Oi Os Out state certs stateInterpret cmd ins outStream.}}{\text{inputTest (inputInterpret x) } \land \\
\text{CFG2Interpret } (M, Oi, Os) \\
\text{CFG2 inputInterpret certInterpret inputTest certs stateInterpret } (x::\text{ins}) \text{ state outStream) } \Rightarrow \\
\text{TR}_{2} (M, Oi, Os) \{ \text{exec (CMD cmd)} \} \\
\text{CFG2 inputInterpret certInterpret inputTest certs stateInterpret } (x::\text{ins}) \text{ state outStream) } \\
\text{CFG2 inputInterpret certInterpret inputTest certs stateInterpret ins } (\text{NS state (exec (CMD cmd)}) \\
\text{(Out state (exec (CMD cmd))::outStream)})} \land \\
\forall \text{inputInterpret certInterpret inputTest x NS M Oi Os Out state certs stateInterpret cmd ins outStream.}}{\text{inputTest (inputInterpret x) } \land \\
\text{CFG2Interpret } (M, Oi, Os) \\
\text{CFG2 inputInterpret certInterpret inputTest certs stateInterpret } (x::\text{ins}) \text{ state outStream) } \Rightarrow \\
\text{TR}_{2} (M, Oi, Os) \{ \text{trap (CMD cmd)} \} \\
\text{CFG2 inputInterpret certInterpret inputTest certs stateInterpret } (x::\text{ins}) \text{ state outStream) } \\
\text{CFG2 inputInterpret certInterpret inputTest certs stateInterpret ins } (\text{NS state (trap (CMD cmd)}) \\
\text{(Out state (trap (CMD cmd))::outStream)})} \land \\
\forall \text{inputInterpret certInterpret inputTest x NS M Oi Os Out state certs stateInterpret cmd ins outStream.}}{\neg \text{inputTest (inputInterpret x) } \Rightarrow \\
\text{TR}_{2} (M, Oi, Os) \text{ discard} \\
\text{CFG2 inputInterpret certInterpret inputTest certs stateInterpret } (x::\text{ins}) \text{ state outStream) } \\
\text{CFG2 inputInterpret certInterpret inputTest certs stateInterpret ins } (\text{NS state discard}) \\
\text{(Out state discard::outStream)})} \Rightarrow \\
\forall a_0 a_1 a_2 a_3. \text{TR}_{2} a_0 a_1 a_2 a_3 \Rightarrow \text{TR}_{2} a_0 a_1 a_2 a_3
\]

\[
\frac{\forall a_0 a_1 a_2 a_3.}{\text{TR}_{2} a_0 a_1 a_2 a_3 \iff \\
\exists \text{inputInterpret certInterpret inputTest x NS M Oi Os Out state certs stateInterpret cmd ins outStream.}}{\langle a_0 = (M, Oi, Os) \rangle \land \langle a_1 = \text{exec (CMD cmd)} \rangle \land \\
\langle a_2 = \\
\text{CFG2 inputInterpret certInterpret inputTest certs stateInterpret } (x::\text{ins}) \text{ state outStream) } \\
\langle a_3 = \\
\text{CFG2 inputInterpret certInterpret inputTest certs stateInterpret ins } (\text{NS state (exec (CMD cmd)}) \\
\text{(Out state (exec (CMD cmd))::outStream)}) \land 
\]
inputTest (inputInterpret x) ∧
CFG2Interpret (M, Oi, Os)

(CFG2 inputInterpret certInterpret inputTest certs
stateInterpret (x::ins) state outStream)) ∨

(∃inputInterpret certInterpret inputTest x NS M Oi Os Out
state certs stateInterpret cmd ins outStream.
(a0 = (M, Oi, Os)) ∧ (a1 = trap (CMD cmd)) ∧
(a2 =
CFG2 inputInterpret certInterpret inputTest certs
stateInterpret (x::ins) state outStream) ∧
(a3 =
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)) ∨

∃inputInterpret certInterpret inputTest x NS M Oi Os Out
state certs stateInterpret cmd ins outStream.
(a0 = (M, Oi, Os)) ∧ (a1 = discard) ∧
(a2 =
CFG2 inputInterpret certInterpret inputTest certs
stateInterpret (x::ins) state outStream) ∧
(a3 =
CFG2 inputInterpret certInterpret inputTest certs
stateInterpret ins (NS state discard)
(Out state discard::outStream)) ∧

¬inputTest (inputInterpret x)

A.4.3 isAuthenticated.def

[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 (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) ∧
\text{(isAuthenticated (v_{10} \text{ says } \text{TT}) \iff F) \land}
\text{(isAuthenticated (v_{10} \text{ says } \text{FF}) \iff F) \land}
\text{(isAuthenticated (Name v_{132} \text{ says } \text{prop } v_{66}) \iff F) \land}
\text{(isAuthenticated (v_{133} \text{ meet } v_{134} \text{ says } \text{prop } v_{66}) \iff F) \land}
\text{(isAuthenticated}
\text{ (Name (Role v_{174}) \text{ quoting } Name (Role v_{164}) \text{ says prop (CMD v_{142})} \iff F) \land}
\text{(isAuthenticated}
\text{ (Name (Key v_{175}) \text{ quoting Name (Role CA) says prop (CMD v_{142})} \iff F) \land}
\text{(isAuthenticated}
\text{ (Name (Key v_{175}) \text{ quoting Name (Role Server) says prop (CMD v_{142})} \iff F) \land}
\text{(isAuthenticated}
\text{ (Name (Key (pubK CA)) \text{ quoting Name (Role (Utility v_{184})) says prop (CMD v_{142})} \iff F) \land}
\text{(isAuthenticated}
\text{ (Name (Key (pubK (Utility v_{190}))) \text{ quoting Name (Role v_{184}) says prop (CMD v_{142})} \iff F) \land}
\text{(isAuthenticated}
\text{ (Name (Key (privK v_{187})) \text{ quoting Name (Role v_{184}) says prop (CMD v_{142})} \iff F) \land}
\text{(isAuthenticated}
\text{ (Name Keyboard \text{ quoting Name (Role v_{164}) says prop (CMD v_{142})} \iff F) \land}
\text{(isAuthenticated}
\text{ (Name (Owner v_{176}) \text{ quoting Name (Role v_{164}) says prop (CMD v_{142})} \iff F) \land}
\text{(isAuthenticated}
\text{ (Name (Account v_{177} v_{178}) \text{ quoting Name (Role v_{164}) says prop (CMD v_{142})} \iff F) \land}
\text{(isAuthenticated}
\text{ (Name v_{154} \text{ quoting Name (Key v_{165}) says prop (CMD v_{142})} \iff F) \land}
\text{(isAuthenticated}
\text{ (Name v_{154} \text{ quoting Name Keyboard says prop (CMD v_{142})} \iff F) \land}
\text{(isAuthenticated}
\text{ (Name (Role v_{192}) \text{ quoting Name (Owner v_{166}) says prop (CMD v_{142})} \iff F) \land}
\text{(isAuthenticated}
\text{ (Name (Key (pubK CA)) \text{ quoting Name (Owner v_{166}) says prop (CMD v_{142})} \iff F) \land}
\text{(isAuthenticated}
\text{ (Name (Key (pubK (Utility v_{206}))) \text{ quoting Name (Owner v_{166}) says prop (CMD v_{142})} \iff F) \land}
\text{(isAuthenticated}
\text{ (Name (Key (privK v_{203})) \text{ quoting Name (Owner v_{166}) says prop (CMD v_{142})} \iff F) \land}
\text{(isAuthenticated}
\text{ (Name (Owner v_{194}) \text{ quoting Name (Owner v_{166}) says}}
\[
\text{prop (CMD \textit{v142}) } \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{Name (Account } \textit{v165} \text{ v166) quoting Name (Owner } \textit{v166} \text{) says} \\
\text{prop (CMD } \textit{v142}) \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{Name } \textit{v154} \text{ quoting Name (Account } \textit{v167} \text{ v168) says} \\
\text{prop (CMD } \textit{v142}) \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v155 meet } \textit{v156} \text{ quoting Name } \textit{v144} \text{ says prop (CMD } \textit{v142}) \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{(v157 quoting } \textit{v158} \text{) quoting Name } \textit{v144} \text{ says} \\
\text{prop (CMD } \textit{v142}) \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v135 quoting } \textit{v145} \text{ meet } \textit{v146} \text{ says prop (CMD } \textit{v142}) \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v135 quoting } \textit{v147} \text{ quoting } \textit{v148} \text{ says prop (CMD } \textit{v142}) \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v135 quoting } \textit{v136} \text{ says prop TRAP) } \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says notf } \textit{v87} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } (\textit{v88 andf } \textit{v99}) \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } (\textit{v70 orf } \textit{v71}) \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } (\textit{v72 impf } \textit{v73}) \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } (\textit{v74 eqf } \textit{v75}) \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } \textit{v76 says } \textit{v77} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } \textit{v78 speaks_for } \textit{v79} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } \textit{v80 controls } \textit{v81} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } \textit{v81 speaks_for } \textit{v82} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } \textit{v82 domi } \textit{v83} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } \textit{v83 eqi } \textit{v84} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } \textit{v84 domi } \textit{v85} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } \textit{v85 eqi } \textit{v86} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } \textit{v86 eqi } \textit{v87} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } \textit{v87 doms } \textit{v88} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } \textit{v89 doms } \textit{v90} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } \textit{v90 eqi } \textit{v91} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } \textit{v91 eqs } \textit{v92} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } \textit{v92 eqn } \textit{v93} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } \textit{v93 lte } \textit{v94} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } \textit{v94 lte } \textit{v95} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } \textit{v95 lte } \textit{v96} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v10 says } \textit{v95 lt } \textit{v96} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v12 speaks_for } \textit{v13} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v14 controls } \textit{v15} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{reps } \textit{v16} \text{ v17 v18} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v19 domi } \textit{v20} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v20 eqi } \textit{v21} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v21 eqs } \textit{v22} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v21 doms } \textit{v23} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v23 eqs } \textit{v24} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v24 eqn } \textit{v25} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v25 eqs } \textit{v26} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v27 eqn } \textit{v28} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v29 lte } \textit{v30} \iff \text{F) } \land \\
(\text{isAuthenticated} \quad \\
(\text{v30 lt } \textit{v31} \iff \text{F) })
References


