Teaching Undergraduates Certified Security by Design

Shiu-Kai Chin, Professor, Electrical Engineering & Computer Science, Syracuse University

Abstract

Design for assurance of security, from the hardware level on up, is essential for securing the integrity of the smart cyber-physical infrastructure that is the Internet of Things. If the smart cyber-physical infrastructure fails to do the right things—that is, if it loses integrity because it is insecure and vulnerable—then untold social consequences will occur. For the security and integrity of cyber-physical systems to improve, not only must engineers and computer scientists possess the capability to design-in security from the very beginning, but they must do so in ways that enable people other than the designers to reproduce and check verification results easily and quickly. Designers and certifiers must formally describe and verify operations at high levels, such as the command-and-control (C2) protocols used by commanders and operators, down to the operations of applications and hardware. We call this design and verification capability for security and integrity *certified security by design* (CSBD). Our experience leads us to conclude that CSBD is feasible and practical for undergraduates. What makes CSBD feasible at the undergraduate level is similar to what made very large scale integrated (VLSI) circuit design feasible in the 1980s: (1) rigorous, simplified, and parameterized design and analytical methods spanning multiple levels of abstraction, and (2) computer-aided design and verification tools to mitigate complexity and problems of scale.

Keywords: assurance**,** computer security, formal verification, integrity, undergraduate education

1. Introduction

"It is not easy to make a computer system secure, but neither is it impossible. The greatest error is to ignore the problem."– Roger Schell (Schell, 1979)

Computer engineering and computer science faculty are responsible for educating the next generation of engineers and scientists capable of routinely producing the next generation of trustworthy cyber-physical systems upon which society depends. The BS degrees in computer engineering and computer science define the baseline capabilities of the computer engineering and computer science professions. For assured security and integrity to permeate cyber-physical systems and the Internet of Things (IoT), undergraduate programs must routinely teach secure system design and verification.

This paper describes an educational approach to *certified security by design* (CSBD) at the *undergraduate* level. Our hope is others will use, modify, or extend our methods so that the number of engineers and computer scientists capable of designing, verifying, implementing, and procuring secure systems will increase dramatically to meet the growing need for secure systems.

As Roger Schell notes (Schell, 1979), designing secure systems is hard but not impossible. Despite the daunting appearance of having to describe and verify secure systems from C2 protocols down to hardware, as educators we cannot perpetuate the continuing insecurity of critical infrastructure by failing to equip future engineers and computer scientists with the capabilities they need to design secure systems.

This is not the first time our profession has risen to the challenge of integrated system design spanning multiple levels of abstraction from high-level descriptions of behavior down to low-level implementations. Incorporating very large scale integrated (VLSI) system design into academic programs in the 1980s was a huge success. Doing so created the technology and engineering workforce necessary for today's IoT. Our experience is that CSBD is feasible within current programs of study in engineering and computer science.

The rest of this paper is organized as follows. Section 2 gives an overview of the historical lessons of the VLSI revolution. Section 3 outlines our approach to CSBD with the lessons of VLSI in mind. Section 4 describes lessons learned from teaching undergraduates. We conclude in Section 5.

2. Historical Lessons

“The transparency of the new methods enabled architects to design systems from top-to-bottom.”
– Lynn Conway (Conway, 2012), page 15

The VLSI Revolution: In the 1970s, the idea that designers working at the instruction-set architecture (ISA) level could realize their designs as custom VLSI circuits was viewed as practically impossible. Circuit design at the semiconductor level involved specialized knowledge of transistors, layouts, and sets of minimum dimensions for layout features, e.g., minimum sizes of transistors and metal interconnects. The combined amount of knowledge needed to do digital design spanning the architecture, logic, transistor, and layout levels was viewed as too much for a single person to grasp.

Forty years later, undergraduate engineering and computer science students routinely span design at the architecture, logic, transistor, and layout levels in their classes. How and why this happened is instructive for those interested in designing secure systems from the hardware level up to and including the missions and applications they support.

Lynn Conway and Carver Mead are credited for the VLSI system design methodology that enables today's students to do what was thought to be impossible in the 1970s. Conway's inspiration was Steinmetz's development of AC electrical system design, as reported in (Conway, 2012).

> *“[T]he emerging AC concepts seemed mysterious, even to expert practitioners, who as yet had no formal theories to develop AC technology. Steinmetz had broken the logjam by coalescing mathematical methods and design examples that enabled practicing engineers to routinely design AC electrical systems with predictable results. This starter set of knowledge was sufficient to launch the AC revolution. By applying Steinmetz's principles, practicing engineers spawned a whole new industry.”*

Conway's insight was simplifying design by finding a “minimalist set of methods” spanning system architecture down to circuit layouts, while discarding everything else, (Conway, 2012).

> *“This wasn't about engineering new things; it was about the engineering of new knowledge. My key idea was 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 sufficient to do any digital design – restructuring the levels of abstraction themselves to be appropriate for MOS-LSI [metal oxide semiconductor-large scale integration].*

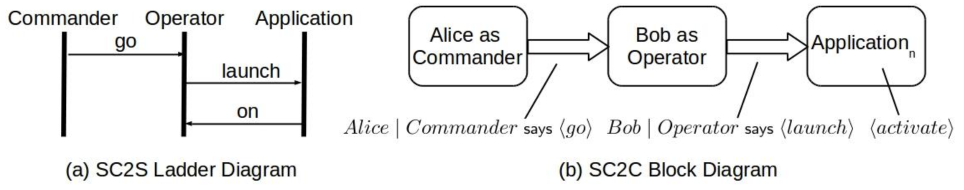


Figure 1: A Simple Command-and-Control Scenario (SC2S)

I theorized that if such a starter set could be composed, it would enable thousands of system designers to quickly migrate from TTL [transistor-transistor logic] into MOS-LSI … We should … create a simplified methodology for designing whole systems in silicon, not just circuits – and aim it specifically at computer architects and system designers."

The simplifications introduced by Conway enabled computer architects to do meaningful design at lower levels of abstraction down to the layout level while preserving logical rigor.

Lessons Learned for Certified Security by Design from the VLSI revolution are these:

1. *Focus on simplifications that enable security to be described and preserved across multiple levels of abstraction*. (a) Coalesce mathematical methods and design examples to enable practicing engineers to routinely design secure systems with predictable results. (b) Attain simplicity and rigor by formally describing and defining the ideal behavior of components. For Conway it was transistors. For security, it includes cryptographic operations. (c) Use parameterization as a means to achieve simplicity and time durability.
2. *Computer-Aided Design (CAD) tools are essential for realizing any security by design methodology*.

With these lessons in mind, next is an overview of our CSBD approach.

3. Certified Security by Design

"Every access to every object must be checked for authority... A foolproof method of identifying the source of every request must be devised."– Jerome Saltzer and Michael Schroeder (Saltzer & Schroeder, 1975)

A Motivating Command-and-Control Example: To illustrate our CSBD approach and the requirements for security assurance, we work through a simple command-and-control scenario (SC2S) as shown in Figure 1. The top-level description of the SC2S C2 protocol is a ladder diagram showing the sequence of commands and messages sent among the principals *Commander*, *Operator*, and *Application* in Figure 1(a). The *Commander* gives the *go* command; the *Operator* in response issues the *launch* command to the *Application*; the *Application* confirms it is *active* by responding with *on* to the *Operator*.

Figure 1(b), with the following description, refines the top-level description and gives additional details related to authentication and authorization.

1. *Alice*, in the role of *Commander*, gives the *go* command to *Bob*, who functions as an *Operator*.
2. *Bob* recognizes that the *Commander* has the authority to give the *go* command. He also knows that *Alice* is the *Commander*. The policy given to *Operators* is when they receive the *go* command, they should *launch* the application for which they are responsible. In Bob's case it is *Application*$_n$.
3. When *Application*$_n$ receives a *launch* command from an *Operator*, in this case *Bob*, it activates, i.e., moves into the *active* state.
4. To secure integrity, cryptographic operations, such as digital signatures, are used to authenticate all commands. *Principals*, such as staff, are associated with cryptographic keys.

Except for the reference to the use of cryptographic operations and digital signatures, the means for authentication and authorization are not explicitly defined.

Assurance Requirements: Even though the SC2S is a simple example, it requires the following tools and methods to maintain a formal and verifiable thread of consistency, accountability, and security to assure (1) all commands are authenticated and authorized, and (2) all actions are justified.

- *A command-and-control (C2) calculus* for describing and justifying actions, commands, or statements made within the context of jurisdiction of authority, delegations, tokens or symbols of identity or authority, policies, privileges, certifications (i.e., signed statements), and trust assumptions (e.g., root cryptographic keys, assumed jurisdictions). We use a calculus (S. Chin &

Older, 2010) based on an access-control calculus for distributed systems (Abadi, Burrows, Lampson, & Plotkin, 1993).

- Computer-assisted reasoning tools to check all proofs and assurance claims, and enable rapid reproduction and formal verification of all results by third parties and certifiers. We use the Cambridge University Higher Order Logic (HOL) theorem prover (Gordon & Melham, 1993).
- *A library of cryptographic operations and their properties* to reason about authentication
- *A method to rigorously and formally account for authentication and authorization as part of the basic foundation of transition systems in general and state machines in particular*. For reasons of scalability, flexibility, and generality, this method must (1) *be fully parameterized* in terms of next-state transition functions, output functions, authentication functions, and authorizations; and (2) support an arbitrarily large number of states, inputs, and outputs.

In the following paragraphs, we give an example solution of each SC2S assurance requirement. Note: in everything that follows, all formulas starting with a "turnstile" ⊢ are *theorems in HOL*, typeset in *LaTeX* by HOL, and formally verified in HOL.

A Command-and-Control (C2) Calculus for Authentication and Authorization: The purpose of the C2 calculus is to justify the actions taken by principals based on (1) the inputs/commands they receive, and (2) the security context they possess in terms of authorizations, certifications, authorities they recognize and their jurisdictions, and root trust assumptions. Some of the inference rules of our C2 calculus are in Figure 2. The rules are used in the typical way: if the current state of the proof has terms that consistently match the pattern of the assumptions above the horizontal line, then the conclusion below the line may be added to the proof. The rules are guaranteed to be logically sound based on their Kripke semantics (S. Chin & Older, 2010).

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

$$\textit{Modus Ponens}\ \frac{\varphi \quad \varphi \supset \varphi'}{\varphi'} \qquad \textit{Says}\ \frac{\varphi}{P \text{ says } \varphi} \qquad \textit{Controls}\ \frac{P \text{ controls } \varphi \quad P \text{ says } \varphi}{\varphi} \qquad \textit{Derived Speaks For}\ \frac{P \Rightarrow Q \quad P \text{ says } \varphi}{Q \text{ says } \varphi}$$

$$\textit{Reps}\ \frac{Q \text{ controls } \varphi \quad P \text{ reps } Q \text{ on } \varphi \quad P \mid Q \text{ says } \varphi}{\varphi} \qquad \textit{\& Says (1)}\ \frac{P \,\&\, Q \text{ says } \varphi}{P \text{ says } \varphi \wedge Q \text{ says } \varphi} \qquad \textit{\& Says (2)}\ \frac{P \text{ says } \varphi \wedge Q \text{ says } \varphi}{P \,\&\, Q \text{ says } \varphi}$$

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

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

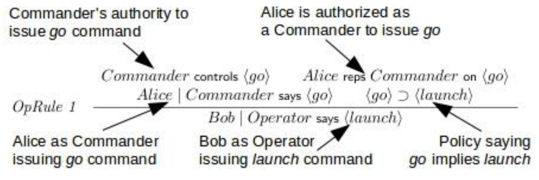
Figure 2: Sample Inference Rules of a Command-and-Control (C2) Calculus

Justifying actions is achieved by taking informal C2 protocol descriptions and formally documenting in the C2 calculus (1) what each principal knows (analogous to the state of a protocol), (2) policies, and (3) received orders. Figure 3(a) lists informal C2 protocol statements and Figure 3(b) shows an inference rule *OpRule 1* derived in the C2 calculus justifying Bob's actions as an operator to issue a *launch* command, as shown in SC2S Figure 1.

OpRule 1 states if (1) Commanders have authority to issue *go* commands, (2) Alice is an authorized Commander, (3) Alice as Commander has issued a *go* command, and (4) the operational policy is when *go* is true so is *launch,* then Bob as an Operator is justified in issuing a *launch* command. The logical soundness of

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

(a) Informal CONOPS Statements and Their Formulas


Commander's authority to issue go command
Alice is authorized as a Commander to issue go
Commander controls ⟨go⟩
Alice reps Commander on ⟨go⟩
Alice | Commander says ⟨go⟩
⟨go⟩ ⊃ ⟨launch⟩
OpRule 1
Bob | Operator says ⟨launch⟩
Alice as Commander issuing go command
Bob as Operator issuing launch command
Policy saying go implies launch

(b) Inference Rule Justifying Bob's Actions as an Operator

Figure 3: Informal CONOPS Statements, Formulas, and Inference Rules

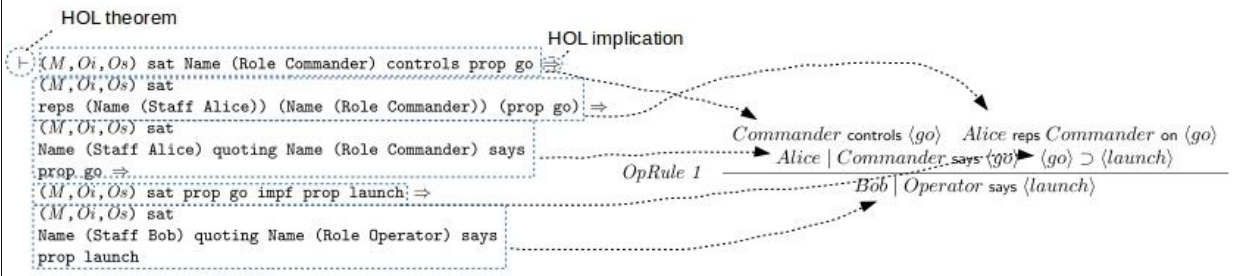
OpRule 1 is proved using the inference rules in Figure 2.

Computer-Assisted Reasoning Tools for Certified Security: The lesson learned from mainstreaming VLSI circuit design is that computer-aided design (CAD) tools are crucial for mitigating complexity and maintaining consistency. For assurance of security, computer-assisted reasoning (CAR) tools are essential for the same reasons. Our CSBD framework uses Cambridge University's Higher Order Logic (HOL) theorem prover (Gordon & Melham, 1993) as a CAR tool. HOL is used because (1) it is higher order, which enables functions to be parameters thus allowing succinct generalizations compared to propositional or first order logics. This avoids state explosion; (2) it is easily extended in a sound fashion by definitional extension and through functions written in the functional language ML; (3) created in 1987, it has a long and proven history of soundness; (4) it has an extensive library of theories; and (5) it was created with hardware verification in mind. For example, it has a formalization of the Acorn RISC Machine (ARM) instruction-set architecture.

The C2 calculus is implemented as a conservative extension to HOL. This means that the logical soundness of HOL is preserved by our implementation of the C2 calculus because the calculus was implemented by a set of definitions and no assumptions about properties were made. Importantly, all properties and theorems of the C2 calculus are proved in HOL.

The benefits of this HOL implementation are (1) correctness of calculated results, (2) human error is reduced to erroneous inputs, which are easily detected, (3) calculation speed, (4) ability to deal with large amounts of detailed and complicated formulas, (5) ability to propagate changes quickly, and (6) most importantly, the ability of certifiers and third parties other than designers to reproduce and check verification results easily and quickly.

Figure 4 shows the HOL theorem corresponding to the OpRule inference rule. In the figure, ⊢ denotes



```
⊢ (M,Oi,Os) sat Name (Role Commander) controls prop go ⇒
  (M,Oi,Os) sat
  reps (Name (Staff Alice)) (Name (Role Commander)) (prop go) ⇒
  (M,Oi,Os) sat
  Name (Staff Alice) quoting Name (Role Commander) says
  prop go ⇒
  (M,Oi,Os) sat prop go impf prop launch ⇒
  (M,Oi,Os) sat
  Name (Staff Bob) quoting Name (Role Operator) says
  prop launch
```

$$\frac{\begin{array}{c} Commander \text{ controls } \langle go \rangle \quad Alice \text{ reps } Commander \text{ on } \langle go \rangle \\ Alice \mid Commander \text{ says } \langle go \rangle \quad \langle go \rangle \supset \langle launch \rangle \end{array}}{Bob \mid Operator \text{ says } \langle launch \rangle} \; OpRule\ 1$$

Figure 4: HOL Theorem Corresponding to OpRule 1

theorems in HOL, => is logical implication in HOL, prop go denotes <*go*>, and prop launch denotes <*launch*>. The prefix *(M, O_i, O_S) sat* in all the terms is due to the Kripke semantics of the C2 calculus.

Modeling Cryptographic Components: Authentication relies on things you know, have, or are, for example, cryptographic keys, identification badges, or biometrics. Cryptographic functions and their properties are important parts of the foundation for security in much the same way as the behavioral properties of transistors are crucial to VLSI design. Any designed-in security approach must include behavioral models of cryptographic operations similar to the switch-level models of transistors. Our CSBD approach includes libraries of cryptographic operations such as cryptographic hashes, asymmetric and symmetric encryption and decryption, and signature generation and verification.

We take advantage of HOL's capability to define algebraic datatypes and prove their properties. For example, we introduce a polymorphic type *digest* corresponding to cryptographic hashes. The *digest* type constructor is *hash* and has the following one-to-one property associated with ideal cryptographic hashes, i.e., two message digests are the same if and only if the messages are identical.

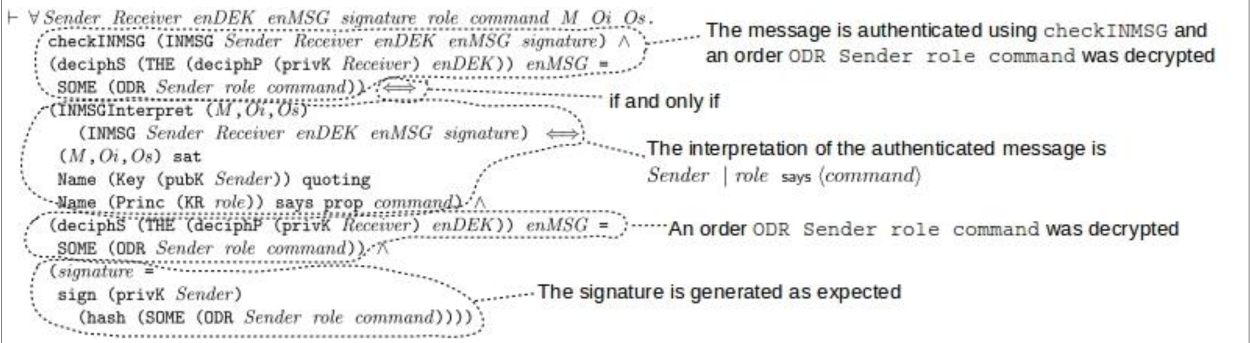
$$\vdash \forall a\ a'.\ (\texttt{hash}\ a = \texttt{hash}\ a') \iff (a = a')$$

In similar fashion, we define algebraic types corresponding to ideal asymmetric and symmetric key encryption and prove their properties. Once encryption is defined, we define the conditions under which the encrypted contents are retrievable. This is shown by the HOL formulas in Figure 5. Signature generation and signature verification are defined in Figure 5(a) and (b), respectively. The properties of *signVerify*, the signature verification function, are given in Figure 6(a). The theorem states that *signVerify* is true if and only if sent and received messages are identical and attributable to the same principal.

Assuring a Unified View of Security: A common understanding of the meaning of inputs, commands, policies, configurations, certificates, and statements, is essential for maintaining operational and security consistency from high-level C2 protocol descriptions down to state-machine transitions. Consistency of interpretation requires defining the meaning of all datatypes and their values. For example, the sequence of three binary digits *111* could mean 7_{10} in unsigned binary arithmetic or -1_{10} in two's-complement arithmetic.

For security, we define the meaning of input messages and state machine configurations in the access-control logic so that we can justify the actions taken in the C2 calculus. As an illustration, recall SC2S. All command-and-control messages have the form

$$INMSG \;\; \text{sender recipient enDEK enInMsg inMsgSig},$$



```
⊢ ∀Sender Receiver enDEK enMSG signature role command M Oi Os.
  checkINMSG (INMSG Sender Receiver enDEK enMSG signature) ∧
  (deciphS (THE (deciphP (privK Receiver) enDEK)) enMSG =
   SOME (ODR Sender role command)) ⟺
  (INMSGInterpret (M,Oi,Os)
     (INMSG Sender Receiver enDEK enMSG signature) ⟺
   (M,Oi,Os) sat
   Name (Key (pubK Sender)) quoting
   Name (Princ (KR role)) says prop command) ∧
  (deciphS (THE (deciphP (privK Receiver) enDEK)) enMSG =
   SOME (ODR Sender role command)) ∧
  (signature =
   sign (privK Sender)
     (hash (SOME (ODR Sender role command))))
```

The message is authenticated using `checkINMSG` and an order `ODR Sender role command` was decrypted

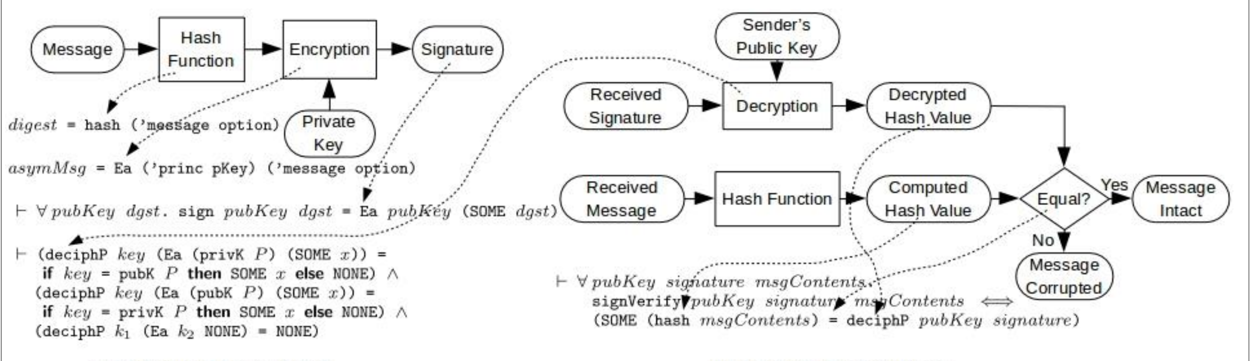
if and only if

The interpretation of the authenticated message is $Sender \mid role$ says $\langle command \rangle$

An order `ODR Sender role command` was decrypted

The signature is generated as expected

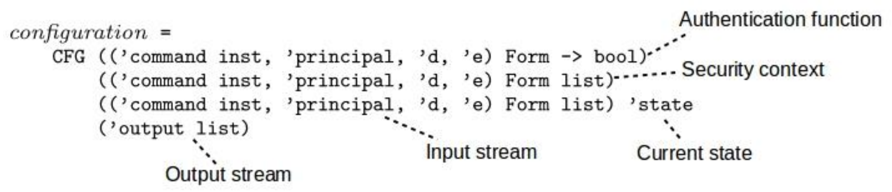
Figure 7: Interpretation of Authenticated Messages in the C2 Calculus



```
digest = hash ('message option)
asymMsg = Ea ('princ pKey) ('message option)
 ⊢ ∀pubKey dgst. sign pubKey dgst = Ea pubKey (SOME dgst)
 ⊢ (deciphP key (Ea (privK P) (SOME x)) =
    if key = pubK P then SOME x else NONE) ∧
   (deciphP key (Ea (pubK P) (SOME x)) =
    if key = privK P then SOME x else NONE) ∧
   (deciphP k₁ (Ea k₂ NONE) = NONE)
```

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

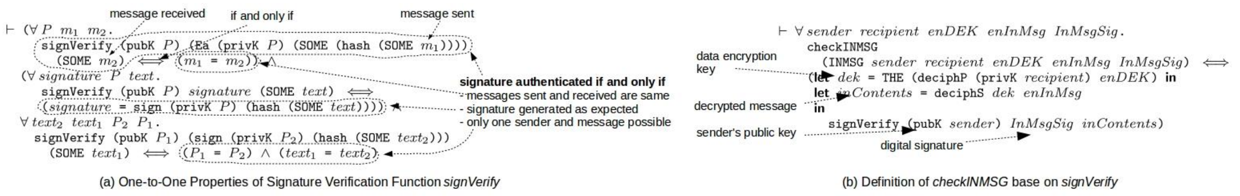
Figure 5: Signature Generation and Verification



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

Figure 8: State Machine Configuration with Authentication and Authorization

where (1) INMSG is viewed as a message header, (2) *sender* is the name of the principal sending the message, (3) *recipient* is the name of the receiver of the message, (4) *enDEK* is the encrypted data encryption key, (5) *enInMsg* is the message symmetrically encrypted using the data encryption key, and (6) *InMsgSig* is the digital signature of the message.



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

message received
if and only if
message sent
signature authenticated if and only if
- messages sent and received are same
- signature generated as expected
- only one sender and message possible

(a) One-to-One Properties of Signature Verification Function *signVerify*

```
⊢ ∀sender recipient enDEK enInMsg InMsgSig.
    checkINMSG
      (INMSG sender recipient enDEK enInMsg InMsgSig) ⟺
    (let dek = THE (deciphP (privK recipient) enDEK) in
     let inContents = deciphS dek enInMsg
     in
       signVerify (pubK sender) InMsgSig inContents)
```

(b) Definition of *checkINMSG* base on *signVerify*

Figure 6: Properties of Signature Verification and Message Authentication Function

C2 messages are defined as HOL datatypes. The *INMSG* authentication function *checkINMSG* is defined as shown in Figure 6(b) using *signVerify*. Its properties are the one-to-one properties of *signVerify*. Figure 7 shows an annotated HOL theorem stating that if a message is authenticated where a recognizable order was successfully decrypted then its meaning in the C2 calculus is *sender | role says <command>*, a recognizable order was decrypted, the signature is as expected, and vice versa. The theorem's importance is it defines the meaning of authenticated messages and the conditions under which the meaning holds.

Parameterized State Machines: State machines are at the foundation of computer hardware. State machine behavior is described by transition relations among machine configurations. Most state-machine descriptions omit security policy as a consideration because either there is no logic or language to describe security policy or security policies are not incorporated into transition relations.

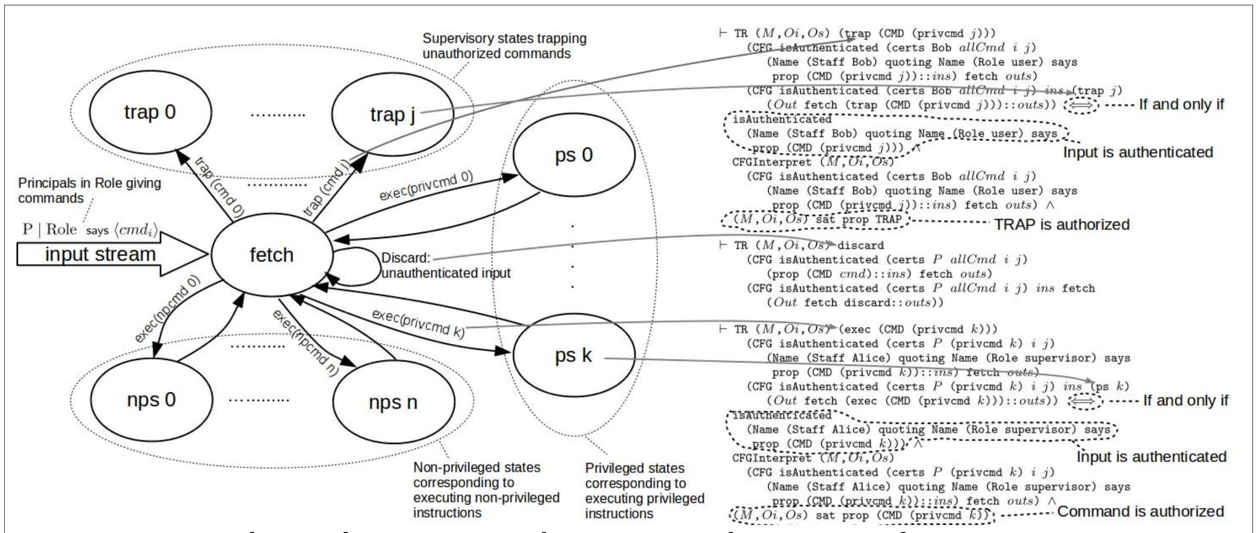
Our CSBD framework incorporates a security context given as an authentication function and a list of certificates, policy statements, and trust assumptions. The security context is a *parameter* of state machine configurations. This parameterized approach allows authentication and authorization to be specialized and changed as required. By including authentication functions and authorization statements that have meaning in the C2 calculus, security is built into the foundation of computer hardware behavioral specifications.

For example, Figure 8 shows the parameters of the HOL datatype *configuration*. Configurations have (1) an authentication function, whose purpose is to identify the source of a command (e.g., by digital signatures or shared secrets); (2) the security context within which authenticated commands are checked for authority (e.g., catalogs of descriptors, capability lists, tickets, mode bits, and access-control lists); (3) an input stream; (4) the current state; and (5) the output stream.

An application of this approach is illustrated by the state machine in Figure 9. Its operation is consistent with Popek's and Goldberg's description of virtualization (Popek & Goldberg, 1974), and Saltzer's and Schroeder's use of a *mode bit* and catalogs of *descriptors* to authorize processes attempting to execute privileged instructions reserved for supervisors (Saltzer & Schroeder, 1975).

Consistent with Popek and Goldberg, we divide commands into two groups: (1) privileged and (2) non-privileged. In a similar fashion, we divide up states into (1) privileged states (which correspond to executing a privileged command), and (2) non-privileged states (which correspond to executing non-privileged commands). Commands and states are indexed by natural numbers, e.g., *npcmd 5* and *privcmd 12* are non-privileged command 5 and privileged command 12, respectively. They are implemented as algebraic datatypes in HOL.

One immediate observation from Figure 9 is our state machine descriptions easily accommodate state machines with arbitrarily large numbers of inputs and states. The use of higher-order logic enables the next



```
⊢ TR (M,Oi,Os) (trap (CMD (privcmd j)))
    (CFG isAuthenticated (certs Bob allCmd i j)
       (Name (Staff Bob) quoting Name (Role user) says
        prop (CMD (privcmd j))::ins) fetch outs)
    (CFG isAuthenticated (certs Bob allCmd i j) ins (trap j)
       (Out fetch (trap (CMD (privcmd j)))::outs)) ⟺
  isAuthenticated
    (Name (Staff Bob) quoting Name (Role user) says
     prop (CMD (privcmd j))) ∧
  CFGInterpret (M,Oi,Os)
    (CFG isAuthenticated (certs Bob allCmd i j)
       (Name (Staff Bob) quoting Name (Role user) says
        prop (CMD (privcmd j))::ins) fetch outs) ∧
  (M,Oi,Os) sat prop TRAP

⊢ TR (M,Oi,Os) discard
    (CFG isAuthenticated (certs P allCmd i j)
       (prop (CMD cmd)::ins) fetch outs)
    (CFG isAuthenticated (certs P allCmd i j) ins fetch
       (Out fetch discard::outs))

⊢ TR (M,Oi,Os) (exec (CMD (privcmd k)))
    (CFG isAuthenticated (certs P (privcmd k) i j)
       (Name (Staff Alice) quoting Name (Role supervisor) says
        prop (CMD (privcmd k))::ins) fetch outs)
    (CFG isAuthenticated (certs P (privcmd k) i j) ins (ps k)
       (Out fetch (exec (CMD (privcmd k)))::outs)) ⟺
  isAuthenticated
    (Name (Staff Alice) quoting Name (Role supervisor) says
     prop (CMD (privcmd k))) ∧
  CFGInterpret (M,Oi,Os)
    (CFG isAuthenticated (certs P (privcmd k) i j)
       (Name (Staff Alice) quoting Name (Role supervisor) says
        prop (CMD (privcmd k))::ins) fetch outs) ∧
  (M,Oi,Os) sat prop (CMD (privcmd k))
```

Figure 9: State-Machine with Instruction Authentication, Authorization, and Trapping

state and output functions to be parameters and avoids the problem of state explosion in our descriptions that occur with model checkers or first-order logic.

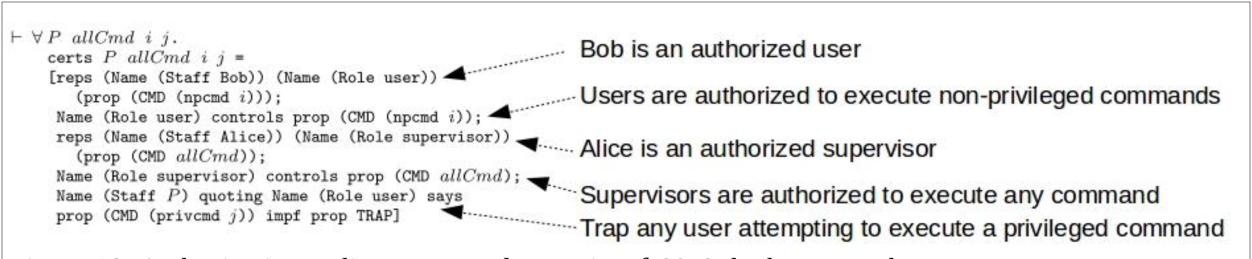
Similar to Saltzer's and Schroeder's mode bit, we define *user* and *supervisor* as the HOL datatype *roles*. As defined in (Popek & Goldberg, 1974), all privileged instructions are trapped if *users* attempt their execution. *Supervisors* are authorized to execute any instruction. Trapping unauthorized instructions is done by a virtual machine monitor (VMM) or hypervisor. Our parameterized state machine of Figure 9 incorporates the functionality of a VMM to trap all unauthorized instruction executions. The theorems in Figure 9 are labeled transitions *TR (M, O_i, O_S) label* relating two configurations if and only if the necessary conditions are true. For example, with respect to *trapping* a user attempting to execute *privcmd j*, the corresponding theorem in Figure 9 has the following form, where the labeled transition relation *TR (M, Oi, Os) trap (CMD (privcmd j))* is represented as $\xrightarrow{trap(CMD(privcmd\ j))}_{(M,O_i,O_s)}$. Transitions from one configuration to another occur if and only if the input is authenticated, a particular security interpretation is used, and the action taken follows from the security interpretation.

$$(configuration_1 \xrightarrow{trap(CMD\ (privcmd\ j))}_{(M,O_i,O_s)} configuration_2) \Longleftrightarrow$$
$$(isAuthenticated(input) \land (CFGInterpret\ (M,O_i,O_s)\ configuration_1) \land (M,O_i,O_s)\ sat\ prop\ TRAP)$$

What the theorem in Figure 9 states is that a *trap(CMD(privcmd j))* transition from *configuration*$_1$ to *configuration*$_2$ occurs if and only if (1) the *input* is authenticated, (2) we are interpreting the security condition of *configuration*$_1$ using *CFGInterpret* and Kripke structure *(M, O*$_i$*, O*$_S$*)*, and (3) *trapping* is justified in the C2 calculus as indicated by *(M,O*$_i$*,O*$_S$*) sat prop TRAP*. The *if and only if* form of this theorem assures that (1) if a trap occurred it is justified by policy and (2) if a trap is justified by policy then it occurs.

The policy is given by parameter *certs p allCMD i j*, whose definition is shown in Figure 10, as a list of access-control logic formulas. Different authentication and authorization policies are implemented by changing the authentication function and security context in configurations.

The definition of *CFGInterpret* is given in Figure 11 as the conjunction of all the access-control logic formulas that are in the policy and authenticated inputs. Shown as well in Figure 11 is the theorem that

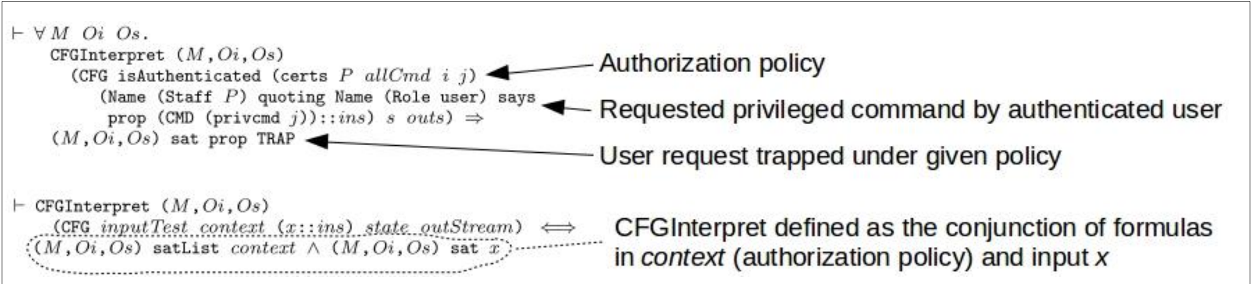


```
⊢ ∀P allCmd i j.
    certs P allCmd i j =
    [reps (Name (Staff Bob)) (Name (Role user))
       (prop (CMD (npcmd i)));
     Name (Role user) controls prop (CMD (npcmd i));
     reps (Name (Staff Alice)) (Name (Role supervisor))
       (prop (CMD allCmd));
     Name (Role supervisor) controls prop (CMD allCmd);
     Name (Staff P) quoting Name (Role user) says
     prop (CMD (privcmd j)) impf prop TRAP]
```

Bob is an authorized user
Users are authorized to execute non-privileged commands
Alice is an authorized supervisor
Supervisors are authorized to execute any command
Trap any user attempting to execute a privileged command

Figure 10: Authorization Policy Expressed as a List of C2 Calculus Formulas

under the security policy interpretation of *configuration*$_1$ a *TRAP* is justified.



```
⊢ ∀M Oi Os.
    CFGInterpret (M,Oi,Os)
      (CFG isAuthenticated (certs P allCmd i j)
         (Name (Staff P) quoting Name (Role user) says
          prop (CMD (privcmd j))::ins) s outs) ⇒
    (M,Oi,Os) sat prop TRAP

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

Authorization policy
Requested privileged command by authenticated user
User request trapped under given policy
CFGInterpret defined as the conjunction of formulas in context (authorization policy) and input x

Figure 11: Theorems Defining Security Interpretation and Trap Justification

4. Lessons Learned Teaching Undergraduates

"Technology is nothing. What's important is that you have a faith in people, that they're basically good and smart, and if you give them tools, they'll do wonderful things with them." – Steve Jobs

What we have described in this paper is what we have taught to undergraduate engineers and computer scientists from over fifty universities in the US and Europe (S.-K. Chin, Devendorf, Muccio, Older, & Royer, 2012). The biggest lesson we have learned from our experience is this: it is reasonable to equip undergraduates with the capabilities to do certified security by design based on formal logic and theorem proving.

The practical lessons we have learned mirror the lessons learned from the VLSI revolution in education.

Focus on simplifications that enable security to be described and preserved across multiple levels of abstraction. Early in our exploration of secure system design, we adopted and soundly modified the access-control calculus for distributed systems described in (Abadi et al., 1993). The C2 calculus used by both undergraduate and graduate students is described fully in (S. Chin & Older, 2010). In the subsequent years after this decision, we have used the C2 calculus to describe operations at the level of institutional/organizational policies (Benson, Chin, Croston, Jayaraman, & Older, 2014), down through communication, authentication, and authorization protocols (Shiu-Kai Chin, Muccio, Older, & Vestal, 2010), to finite-state machines. The background necessary to learn the C2 calculus is undergraduate discrete mathematics. The C2 calculus enables students to rigorously connect operational descriptions across multiple levels of abstraction. Most importantly, the C2 calculus enables students to precisely and accurately describe authentication and authorization in a vertically integrated way, linking high-level descriptions to implementations.

Define and prove properties of ideal cryptographic components. Simple algebraic models of cryptographic operations, such as cryptographic hashes, and symmetric and asymmetric encryption are useful and necessary for linking C2 protocols and their description in the C2 calculus with the structure and interpretation of C2 messages and certificates used in C2 protocols. The algebraic models are general in nature with an array of useful properties expressed as HOL theorems. The definitions and theorems of cryptographic operations are a useful library for students using cryptographic components in their systems. When we made these libraries available to students, they were able to use them soundly within the HOL theorem prover. We also found that theorems where logical equivalence is the top-level operator instead of logical implication are much easier for students to use. This is because rewriting is simpler than resolution in HOL.

Parameterization and higher-order functions provide simplicity and time durability. Embedding the C2 calculus within higher order logic and HOL is essential for combining security descriptions with behavioral descriptions using state machines, for example. Leveraging the capability to parameterize functions such as next-state and output functions greatly simplifies and generalizes system descriptions. Such generalization is the key for proving properties that are parameterized theorems applicable to all state machines.

CAR tools are essential for realizing any security by design methodology. Just as CAD tools and libraries of components provide a sound platform for students designing VLSI systems, CAR tools and theory libraries provide a logically sound platform for students designing systems with assured security and integrity. Students commented often that using HOL sharpened their understanding of security and the systems they were assuring. Proving theorems in HOL gives them the confidence that their theorems really are sound. More importantly, using HOL allows others with more experience to have confidence in the results produced by less experienced students and engineers.

We used HOL because of its high-order nature, long-term reliability and extensibility, and its voluminous theory library. Our results should be portable to other higher-order CAR tools.

Students with functional programming skills have an easier time. Theorem provers such as HOL are operated through functional languages such as ML. Security properties are often expressed as properties of functions. Students who are comfortable with and capable users of strongly-typed functional languages have a significantly lower learning curve compared to students who only know imperative programming languages. However, it is possible for students without prior functional programming language experience and expertise to pick up these skills while learning ML and HOL.

5. Conclusion

"You cannot escape the responsibility of tomorrow by evading it today."– Abraham Lincoln

As educators of the builders of the next generation of systems, it is our responsibility to equip the next generation of builders with the capabilities and tools they need to design secure systems. We have found that students are eager to meet the challenge to build better systems with integrity and security built-in from the start. As modern society becomes ever more dependent on cyber-physical systems and the IoT, our obligation to future generations is to give them the engineers and computer scientists capable of securing that future. Certified security by design shows that we can meet our obligation to the future routinely without heroics.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1245867; by Serco, Inc., through subcontract number COS1000687 under US Air Force Research Laboratory Rome contract FA8750-10-C-00116. Additional support came from US Air Force Research Laboratory Rome contract FA8750-12-1-0214. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the US Air Force or the US Government. Dr. Patricia Lowney provided many useful comments, refinements, and improvements to the ideas and text in this paper.

References

Abadi, M., Burrows, M., Lampson, B., & Plotkin, G. (1993). A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems, 15*(4), 706-34. doi:10.1145/155183.155225

Benson, G., Chin, S.-K., Croston, S., Jayaraman, K., & Older, S. (2014). Banking on interoperability: Secure, interoperable credential management. *Computer Networks, 67*, 235-251. doi:10.1016/j.comnet.2014.03.024

Chin, S.-K., Devendorf, E., Muccio, S., Older, S., & Royer, J. (2012). *Formal verification for mission assurance in cyberspace: Education, tools, and results*. *Proceedings 16Th Colloquium for Information Systems Security Education,* 75-82.

Chin, S.-K., & Older, S. B. (2010). *Access control, security, and trust: A logical approach* (1st Ed.) CRC Press.

Conway, L. (2012). Reminiscences of the VLSI revolution: How a series of failures triggered a paradigm shift in digital design. *IEEE Solid-State Circuits Magazine, 4*(4), 8-31. doi:10.1109/MSSC.2012.2215752

Gordon, M. J. C., & Melham, T. F. (1993). *Introduction to HOL: A theorem proving environment for higher order logic* Cambridge University Press.

Popek, G. J., & Goldberg, R. P. (1974). Formal requirements for virtualizable third generation architectures. *Communications of the ACM, 17*(7), 412-21. doi:10.1145/361011.361073

Saltzer, J. H., & Schroeder, M. D. (1975). The protection of information in computer systems. *Proceedings of the IEEE, 63*(9), 1278-1308. doi:10.1109/PROC.1975.9939

Schell, R. R. (1979). Computer security: The achilles' heel of the electronic air force? *Air University Review, 30*(2), 16-33.

Shiu-Kai Chin, Muccio, S., Older, S., & Vestal, T. N. J. (2010). Policy-based design and verification for mission assurance. *5th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security, MMM-ACNS 2010,* 125-38. doi:10.1007/978-3-642-14706-7_10