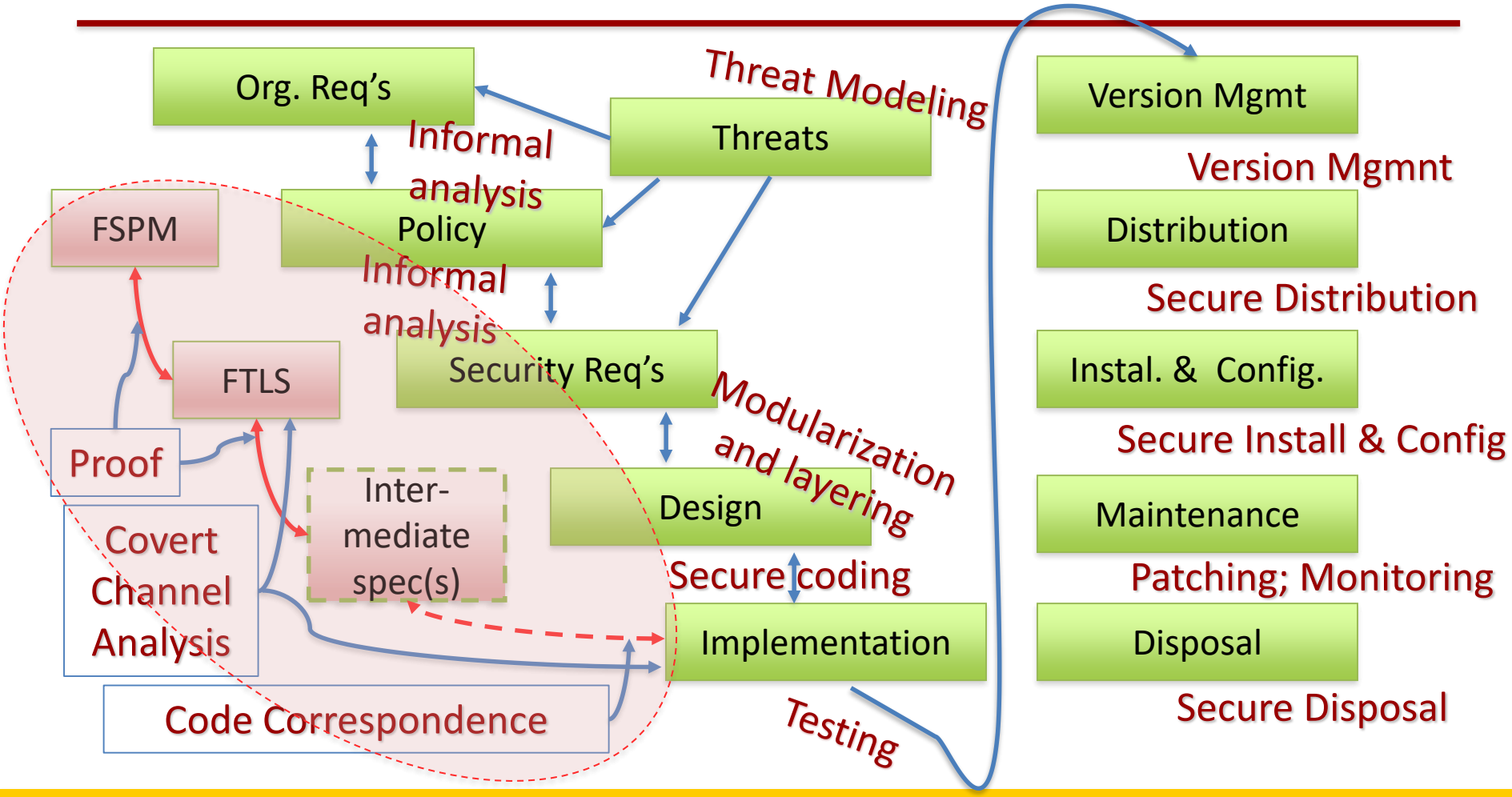
INF523: Computer System Assurance

Formal Methods

Prof. Clifford Neuman

Lecture 9
23 October 2020

"Assurance Waterfall"



Reading for (this) Time

- Jonathan K. Millen. 1976. Security Kernel validation in practice. *Comm. ACM* 19, 5 (May 1976), 243-250. DOI=10.1145/360051.360059
- T. Levin, S. Padilla, and R. Schell, Engineering Results from the A1 Formal Verification Process, in *Proceedings of the 12th National Computer Security Conference*, Baltimore, Maryland, 1989. pp. 65-74

More Reading for This Time

- Bishop, pp. 545-551
  - A Specifier's Introduction to Formal Methods, Jeannette M. Wing
  - Formal Specifications, a Roadmap, Axel van Lamsweerde

Formal Methods

- *Formal* means *mathematical*
- Tools and methods for reasoning about correctness
  - *Correctness* means system design satisfies some properties
  - Security, but also safety and other types of properties
- Useful way to think completely, precisely, and unambiguously about the system
  - Help delimit boundary between system and environment
  - Characterize system behavior under different conditions
  - Identify assumptions
  - Identify necessary invariant properties
- Often find flaws just from writing formal specification

Informal vs. Formal Specifications

- Informal
  - Human language, descriptive
  - E.g., "The value of variable *x* will always be less than 5"
  - Often vague, ambiguous, self-contradictory, incomplete, imprecise, and doesn't handle abstractions well
    - **All of which can easily lead to unknown flaws**
  - But, relatively easy to write
- Formal
  - Mathematical
  - E.g., $\forall t. \forall x. (t >= x \wedge (sys\_init(x))) \rightarrow x(t) < 5$
  - Easily handles abstractions, concise, non-ambiguous, precise, complete, etc.
  - But, requires lots of training and experience to do right

Always? What about before the system is (re)initialized?

Formal vs. “Informal” Verification

- “Informal” verification:
  - **Testing** of various sorts
    - Finite, can never can be complete, only demonstrates cases
- Formal verification:
  - Application of formal methods to “prove” a design satisfies some requirements (properties)
    - A.k.a. “demonstrating correctness”
  - Can “prove” a system is secure
    - I.e., that the system design satisfies some properties that are the definition of “security” for the system
    - I.e., that a system satisfies the security policy

Some Uses of Formal Methods

- Prove certain properties
  - E.g., invariants, such as BLP always in secure state
- Prove that certain combinations of states never occur
- Prove value of certain variable never exceeds bounds
- Prove absence of information flows
  - E.g., for transitive closure of shared resource matrix

- Very widely used for hardware
- Not currently widely used for software

Types of Formal Verification

- Theorem proving (semi-automated)
  - Proving of mathematical theorems
    - E.g., that FTLS satisfies FSPM
  - Complex, prone to error if done totally by hand
  - Must use automated (mechanized) theorem proving tools
    - Can solve some simple proofs automatically using heuristics
    - Non-trivial proofs require lots of human input
- Model checking (automated)
  - Specify system as FSM, properties as valid states
    - Exhaustively compare possible system states to specification to show all states satisfy spec
  - May run a long time for complex state
    - Use heuristics in advance to prune state space

Steps in Security Formal Verification

1. Develop FSPM (e.g., BLP)
2. Develop Formal Top-Level Spec (FTLS)
   - Contrast with Descriptive Top-Level Specification (DTLS)
     - Natural language, not mathematical, specification
3. Proof (formal or informal) that FTLP satisfies FSPM
4. (Possibly intermediate specs and proofs)
   - At different levels of abstraction
5. Show implementation "corresponds" to FTLS
   - Code proof beyond state of the art (but see https://sel4.systems/)
   - Generally informal arguments
   - Must show how every part of code fits

Attributes of Formal Specifications

- States what system does, but not how
  - I.e., like module interfaces from earlier this semester
  - Module interfaces are (probably informal) specifications
- Precise and complete definition of effects
  - Effects on system state
  - Results returned to callers
  - All side-effects, if any
- Not the details of *how*
  - Not how the data is stored, etc.
  - I.e., abstraction
- Formal specification language is not code

Parts of a Formal Specification

- Basic types of entities
  - E.g., in BLP, subjects and objects, access modes
- State variables
  - E.g., b, M, f, and H
- Defined concepts and relations
  - In terms of entities and state variables
  - E.g., dominance, SSC, *-property
- Operations
  - E.g., get_read
  - Relations of inputs to outputs – e.g., R, D, W
  - State changes

Bell-La Padula Formal Policy Model

- From "Secure Computer System: Unified Exposition and Multics Interpretation", Appendix

Rule 1 (R1): get-read

Domain of R1: all $R_k = (g, S_i, O_j, r)$ in $R^{(1)}$. (Denote domain of R_i by dom (R_i).)

Semantics: Subject S_i requests access to object O_j in read-only mode (r).

*-property function: $*1\ (R_k, v) = \text{TRUE} \Leftrightarrow f_c(S_i) \propto f_o(O_j)$.

The rule:

$$R1(R_k,v) = \begin{cases} (?, v) & \text{if } R_k \notin \text{dom } (R1); \\ (\text{yes}, (b \cup (S_i,O_j,r)^{\dagger}, M, f, H)) & \text{if } [R_k \in \text{dom } (R1)] \ \& \ [r \in M_{ij}] \ \& \ [f_s(S_i) \propto f_o(O_j)] \ \& \ [S_i \in S_T \text{ or } *1(R_k,v)]; \\ (\text{no}, v) & \text{otherwise.} \end{cases}$$

New state

Error if invalid call or not a get_read call, and no change to state

Discretionary and mandatory policy requirements

If valid get_read call but does not satisfy discretionary or mandatory policy, no change to state

Algorithm for R1:

if $R_k \notin$ dom (R1) then $R1(R_k, v) = (?, v)$; else if $r \in M_{ij}$ and $\langle [S_i \in S' \text{ and } *1(R_k, v)] \text{ or } [S_i \in S_T \text{ and } f_s(S_i) \propto f_o(O_j)] \rangle$
then $R1(R_k, v) = (\text{yes}, (b \cup (S_i, O_j, r), M, f, H))$;
else $R1(R_k, v) = (\text{no}, v)$;

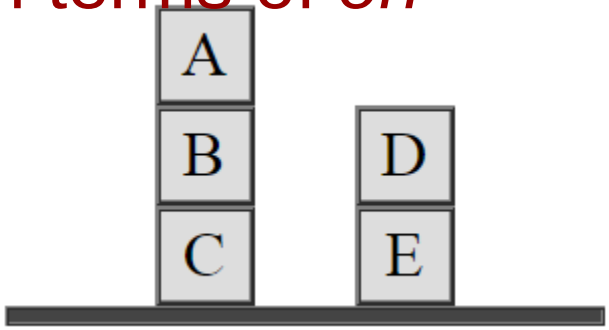
end;

Formal Top-Level Specification

- Represents interface of the system
  - In terms of exceptions, error messages, and effects
  - Must be shown to accurately reflect TCB interface
  - Include HW/FW operations, if affect state at interface
  - TCB “instruction set” consists of HW instructions accessible at interface and TCB calls
- Describe external behavior of the system
  - precisely,
  - unambiguously, and
  - in a way amenable to computer processing for analysis
  - Without describing or constraining implementation

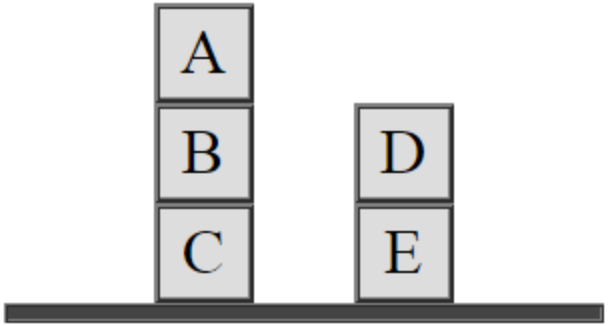
Creating a Formal Specification

- Example, “blocks world”
- 5 objects *{a,b,c,d,e}*
  - Table is not an object in this example
- Relations *{on,above,stack,clear,ontable}*
- *on(a,b); on(b,c); on(d,e)*
- ¬*on(a,a)*, ¬*on(b,a)*, … , etc.
- Define all of the other relations in terms of *on*



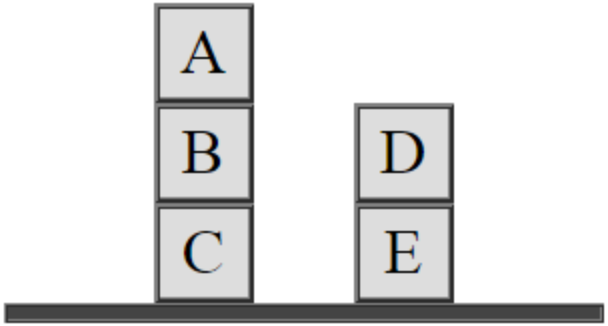
Creating a Formal Specification

- Define all of the other relations in terms of *on*
- ∀y.(clear(y) ⇔
  ¬∃x.on(x,y))
- ∀*x*.(*ontable*(*x*) ⇔
  ¬∃*y*.*on*(*x*,*y*))
- ∀x.∀y.∀z.(stack(x,y,z) ⇔
  on(x,y) ∧ on(y,z))
- ∀*x*.∀*z*.(*above*(*x*,*z*) ⇔
  on(*x*,*z*) ∨ ∃*y*.(*on*(*x*,*y*) ∧ *above*(*y*,*z*)))
  - We are missing something for *above*. What is it?
    ∀x.¬above(x,x)



Alternative Specification

- Define all of the other relations in terms of *above*
- ∀x.(ontable(x) ⇔ ¬∃y.above(x,y))
- ∀x.(clear(x) ⇔ ¬∃y.above(y,x))
- ∀x.∀y.(on(x,y) ⇔
  above(x,y) ∧ ¬∃z.(above(x,z) ∧
  above(z,y))
- What about *stack*?
  - Can define in terms of on, as before
- Need other axioms about *above*:
- ∀x.¬above(x,x)
- ∀x.∀y.∀z. above(x,y) ∧ above(y,z) => above(x,z)
- ∀x.∀y.∀z. above(x,y) ∧ above(x,z) =>
  y=z \/ above(y,z) \/ above(z,y)
- ∀x.∀y.∀z. above(y,x) ∧ above(z,x) =>
  y=z \/ above(y,z) \/ above(z,y)



Observation

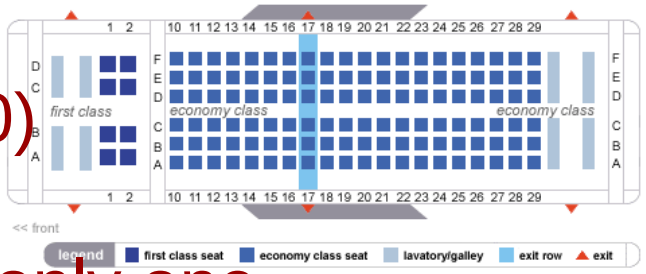
- Many ways to specify the same system
- Not every way is equally good
- If pick less good way, may create lots of complexity
- E.g., consider how to specify a FIFO queue
  1. Infinite array with index of current head and tail
     - Not very abstract – specifies "how"
  2. Simple, recursive, add and remove functions and axioms
     - E.g., ∀x. remove(add(x,EMPTY)) = x
- The first is tedious to reason with
  - Lots of "overhead" to keep track of indexes
- The second is easy and highly automatable

Formal System Specifications

- Previous example used first-order logic (FOL)
  - ∀ and ∃
- For complex systems, FOL may not be enough
- Want "higher-order" logic (HOL), which can take functions as arguments
- E.g., [Rushby PVS phone book example]
  - http://www.csl.sri.com/papers/wift-tutorial/slides.pdf

Homework (due 11/2/2020)

- Write a formal spec for seating in an airplane:



- An airplane has 100 seats (1..100)
- Every passenger gets one seat
- Any seat with a passenger holds only one passenger
- The state of a plane P is a function [N -> S]
  - Maps a passenger name to a seat number
- Two functions: assign_seat and deassign_seat
- Define the functions
- Show some lemmas that demonstrate correctness

Start of Homework Solution

- Types:
  - N : type (of passenger)
  - S : type (of seat number)
  - A : type (of airplane function) [N -> S]
  - e0 : N (represents an empty seat)

- Variables:
  - nm : var N (a passenger)
  - pl : var A (an airplane function)
  - st : var S (a seat number)

What you Need to Do

1. Define the axioms for the two functions:
   - assign_seat : [A x N x S -> A]
   - deassign_seat : [A x S -> A]
2. Be careful that the spec covers all requirements:
   - Can someone have “e0” as their seat number?
   - Can a passenger have more than one seat?
   - Can a seat have more than one passenger?
3. Identify some lemmas that demonstrate that the system specification describes what is intended and sketch the proof

Formal Verification is Not Enough

- Formal verification complements, but does not replace testing (informal verification)
- Requires abstraction which
  - May leave out important details (stuff missing)
  - May make assumptions that code does not support (extra stuff)
- Even if “proven correct”, may still not be correct

- “Beware of bugs in the above code; I have only proved it correct, not tried it.” -Knuth

INF523: Assurance in Cyberspace as Applied to Information Security

Case Studies of Formal Specification and Proofs

Reading for This Class

- Jonathan K. Millen. 1976. Security Kernel validation in practice. *Commun. ACM* 19, 5 (May 1976), 243-250. DOI=10.1145/360051.360059
- T. Levin, S. Padilla, and R. Schell, Engineering Results from the A1 Formal Verification Process, in *Proceedings of the 12th National Computer Security Conference*, Baltimore, Maryland, 1989. pp. 65-74

DEC PDP 11

- Sold by DEC
- 1970s-1990s
- Most popular minicomputer ever
- Smallest mini-computer for a decade that could run Unix

Millen: PDP 11/45 Proof of Correctness

- Proof of correctness for PDP 11/45 security kernel
- Correctness defined as proper implementation of security policy model (BLP)
- Security policy model defined as set of axioms
  - Axioms are propositions from which properties are derived
  - E.g., in BLP, SSC and *-property
- Proof is that all operations available at the interface of the system preserve the axioms
- Also considered covert storage channels
  - Method did not address timing channels

Millen: PDP 11/45 Proof of Correctness

- Security policy model defined as set of axioms
  - Simple security condition
    - If a subject has “read” access to an object, level of subject dominates level of object
  - *-property
    - If a subject has “read” access to one object and “write” access to a second object, level of second object dominates level of first object
  - Tranquility principle for object levels
    - Level of active object will not be changed
  - Exclusion of read access to inactive objects
  - Rewriting (scrubbing) of objects that become active

Layers of Specification and Proof

- Four stages
- Each stage more detailed and closer to machine implementation than the one before

1. FSPM (BLP)
2. FTLS – The interface of the system
   - Includes OS calls and
   - PDP 11/45 instructions available outside kernel
3. Algorithmic specification – High-level code that represents machine language
   - Semantics of language must be well-understood
4. Machine itself: Running code and HW

Why Four Proof Stages?

- Simplify proof work
- Big jump from machine to FSPM
  - FSPM has subjects, objects, *-property, …
  - Machine has code and hardware
- Intermediate layers are closer to each other
- First prove FTLS is valid interpretation of FSPM
- Then further proofs only need to show that lower stages implement FTLS
  - Lower-level proofs don't need abstractions of subjects and objects and *-property

Stages 1 and 2 Specification Format

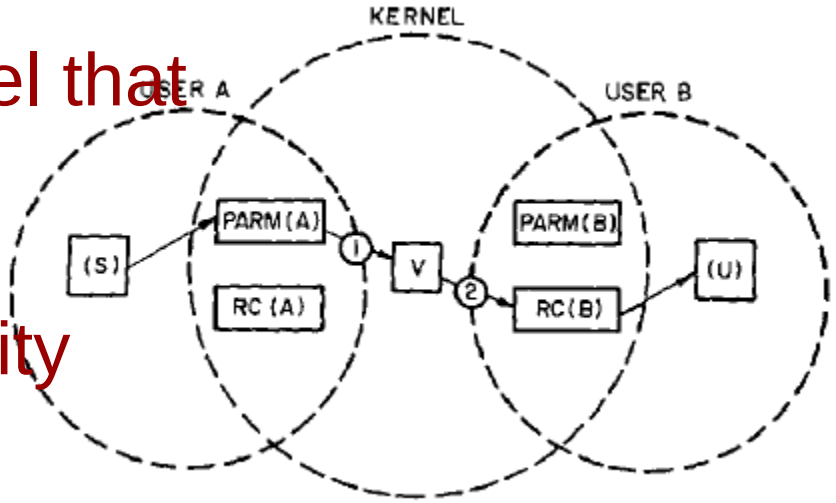
- Both FSPM and FTLS are state machines
  - States and transitions
  - E.g., BLP state is (b, M, f, H)
  - FTLS transitions:
    - Create (activate) object
    - Delete (deactivate) object
    - Get access to an object for a subject
    - Release access to an object for a subject
    - Put a subject in an object's ACL
    - Remove a subject from an object's ACL
    - PDP-11/45 instructions available at interface

V- and O-functions

- State variables and kernel operations are functions
  - State variables are represented as V-functions
    - All V-functions are references to objects
  - Operations are O-functions
    - By subjects to objects
    - Accesses are due to O-function executions
- O-functions have effects on state variables
  - Indicated by values of V-functions before and after
  - E.g., $\neg$`(PS_SEG_INUSE(TCP,dseg))` $\wedge$ `RC(TCP) = NO`
  - I.e., if the object is not in use by the subject then the return code is “NO”

“Shared Resource Problems”

- Covert storage (not timing) channels
- User A at high level
- Modifies kernel state variable V
- User B at low level
- Receives value from kernel that was influenced by V
- Detect by assigning security level to internal variables like V



Proof Example in Paper

- Verification that DELETE enforces *-property
- Original spec at right
- Effect statements are labeled “A”, “B”, etc.
  - Used to simplify statement form for proof
- x = ‘y’ means subject has read access to y and write access to x

```
   Function:  DELETE
   parameters:  dseg,entry
   let:  dpn = PS_SEG(TCP,dseg)
   effect:
A    if not PS_SEG_INUSE(TCP,dseg) or
B       not AST_WAL(dpn,TCP)  or
        AST_TYPE(dpn,TCP) ≠ DIRECTORY  or
          ´DIR_SIZE´(dpn,entry) = 0
C    then RC(TCP) = NO
     else RC(TCP) = YES  and
          DIR_SIZE(dpn,entry) = 0  and
          if ´DIR_ACL_HEAD´(dpn,entry) ≠ 0
D         then ACL_CHAIN(dpn,FINDEND(dpn,entry))
                 = ´ACL_CHAIN´(dpn,0)  and
                 ACL_CHAIN(dpn,0) = ´DIR_ACL_HEAD´(dpn,entry)
          end
          and (¥pn)
E(pn)     if pn ≥ dpn.entry
          then if pn = apn.aentry
                  and DIR_TYPE(apn.aentry) = DIRECTORY
               then (¥bentry) DIR_SIZE(pn,bentry) = 0 and
                    (¥acle) ACL_CHAIN(pn,acle)
F(pn)                        = acle + 1 (mod acle_max + 1)
               else VM(pn) = 0
               end  and
               WAIT_SET(pn) = ∅ and
               SMFR_COUNT(pn) = 1
               and (¥aproc)
```

Abbreviated DELETE Specification

- Statements abstracted to show structure
- Bottom version is in "conjunctive form"
- "Else" sometimes replaced by negation of the "If" condition
- Statements in form **`if f then g else h end`** sometimes converted to **`(f ∧ g) ∨ (¬f ∧ h)`**

```
If A pr B
then C
else D and
          (¥pn)
          if E(pn)
          then F(pn) and
                    (¥aproc)
                    if G(pn,aproc)
                    then H(pn,aproc)
                    end
          end and
          (¥aproc)
          if J(aproc)
          then K(aproc)
          end
end
```

```
          if A then C else if ~A and B then C else D end
and (¥pn) if ~A and ~B and E(pn) then F(pn) end
and (¥aproc)
          if ~A and ~B
          then (¥pn)
                    if E(pn) and G(pn,aproc)
                    then H(pn,aproc)
                    end
          end
and (¥aproc) if ~A and ~B and J(aproc) then K(aproc) end
```

Proof Technique: Security Levels

- Object levels
  - Level based on pathname *pn* of object in hierarchy
  - Level of object at pathname *pn* is *L(pn)*
  - V-functions that take *pn* as parameter have level *L(pn)*
  - Constant V-functions (no parameters) have sys-low level
- Level of subject with process number *proc* is *PL(proc)*
  - *PL(proc)* is level where subject can both read and write
  - V-functions for subjects (i.e., that read state values for processes, as opposed to system state) have level *PL(proc)*
  - O-functions have process numbers as parameters
  - O-functions and their parameters have level *PL(proc)*

Property Cases and Security Levels

```
   Function:  DELETE
   parameters:  dseg,entry
   let:  dpn = PS_SEG(TCP,dseg)
   effect:
A    if not PS_SEG_INUSE(TCP,dseg) or
B       not AST_WAL(dpn,TCP)  or
        AST_TYPE(dpn,TCP) ≠ DIRECTORY  or
          'DIR_SIZE'(dpn,entry) = 0
C    then RC(TCP) = NO
     else RC(TCP) = YES  and
          DIR_SIZE(dpn,entry) = 0  and
          if 'DIR_ACL_HEAD'(dpn,entry) ≠ 0
D         then ACL_CHAIN(dpn,FINDEND(dpn,entry))
                 = 'ACL_CHAIN'(dpn,0)  and
                 ACL_CHAIN(dpn,0) = 'DIR_ACL_HEAD'(dpn,entry)
          end
          and (¥pn)
E(pn)     if pn ≥ dpn.entry
          then if pn = apn.aentry
                  and DIR_TYPE(apn.aentry) = DIRECTORY
               then (¥bentry) DIR_SIZE(pn,bentry) = 0 and
                    (¥acle) ACL_CHAIN(pn,acle)
F(pn)                     = acle + 1 (mod acle_max + 1)
               else VM(pn) = 0
               end  and
               WAIT_SET(pn) = ∅ and
               SMFR_COUNT(pn) = 1
               and (¥aproc)
```

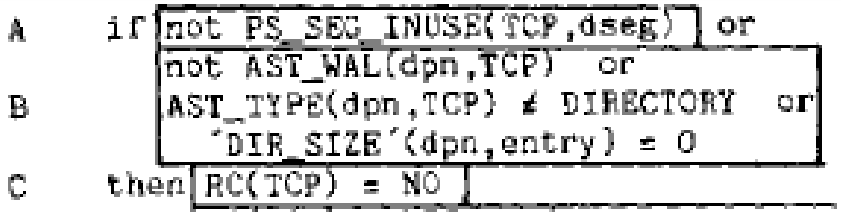
Table I. *-Property Cases.

Subject/ Case	Condi- tions	Ef- fects	Read Levels	Write Levels
S_1/I	A	C	$PL(TCP)$	$PL(TCP)$
S_1/II	$\bar{A}\ B$	C	$PL(TCP), L(dpn)$	$PL(TCP)$
S_1/III	$\bar{A}\ \bar{B}$	D	$PL(TCP), L(dpn)$	$PL(TCP)$, $L(dpn)$
$S_2(pn)$	$\bar{A}\ \bar{B}\ E$	F	$PL(TCP), L(dpn)$, $L(apn)$	$L(pn)$
$S_3(aproc)$	$\bar{A}\ \bar{B}\ E\ G$	H	$PL(TCP), L(dpn)$, $PL(aproc), L(pn)$	$PL(aproc)$
$S_4(aproc)$	$\bar{A}\ \bar{B}\ J$	K	$PL(TCP), L(dpn)$, $PL(aproc)$	$PL(aproc)$

Table II. Security Levels of V-Function References in Labeled Statements.

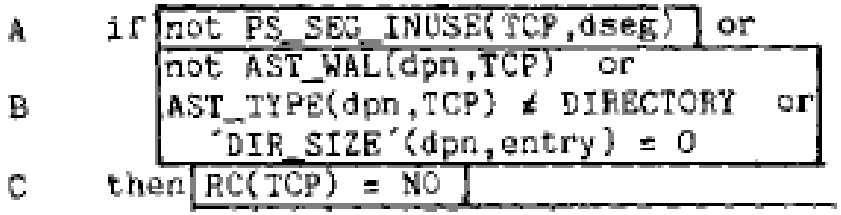
Statement	Read Levels	Write Levels
A	$PL(TCP)$	—
B	$PL(TCP), L(dpn)$	—
C	—	$PL(TCP)$
D	$PL(TCP), L(dpn)$	$PL(TCP), L(dpn)$
E (pn)	$PL(TCP)$	—
F (pn)	$L(apn)$	$L(pn)$
G $(pn, aproc)$	$PL(aproc)$	—
H $(pn, aproc)$	$PL(aproc), L(pn)$	$PL(aproc)$
J $(aproc)$	$PL(aproc)$	—
K $(aproc)$	$PL(aproc)$	$PL(aproc)$

Proof Case Example: Explanation of A, B, C



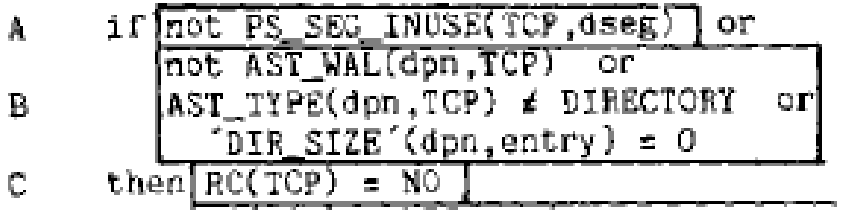
- Delete(dseg,entry)
  - Erases a segment from a directory
  - *Dseg* is directory segment, *entry* is index in directory
- (A) If the local segment number is not in use, or
- (B) If the process does not have write access to the directory or the directory entry is empty
- (C) Return “NO”

Proof Case Example: Function Explanation



- *dpn* is abbreviation for *PS_SEG(TCP,dseg)*
  - Directory path name
- **Active segment table (AST) is part of global state**
- **AST entry numbers must be invisible to avoid channel**
- PS_SEG(TCP,dseg) maps process-local segment numbers to active segment table AST entries
- PS_SEG_INUSE indicates whether or not an element in PS_SEG is in use
- AST_WAL is active segment table write access list

Proof Case Example: Proof Goal



Subject/ Case	Condi- tions	Ef- fects	Read Levels	Write Levels
S_1/I	A	C	$PL(TCP)$	$PL(TCP)$
S_1/II	$\bar{A}\ B$	C	$PL(TCP), L(dpn)$	$PL(TCP)$

- Second case: ¬A ∧ B ∧ C
  - PS_SEG_INUSE(TCP,dseg) = TRUE and
  - AST_WAL(dpn,TCP) = FALSE or… or …
- Prove second case does not violate *-property
- Process is reading from the directory and writing to the response code, so must prove: L(dir) ≤ L(RC)
- I.e., **L(dpn) ≤ PL(TCP)**

Proof Case Example: Proof

Two general relations suffice to complete the proof of the inequality:

R1: $(\forall proc)(\forall seg)$ if $PS\_SEG\_INUSE(proc, seg) = TRUE$
then $AST\_CPL(PS\_SEG(proc,seg), proc) = TRUE$.

R2: $(\forall pn)(\forall proc)$ if $AST\_CPL(pn, proc) = TRUE$
then $L(pn) \leq PL(proc)$.

- R1: If PS_SEG_INUSE is true then it must be the case that the process is in the AST "connected process list" (CPL) for that segment
- R2: If a process is in the AST_CPL then it must be the case that **`L(pn) ≤ PL(proc)`**
- Relations proven inductively over all operations

GEMSOS Verification

- PDP 11/45 verification before TCSEC
- GEMSOS developed to meet TCSEC class-A1
- Gemini Trusted Network Processer (GTNP) developed to be TNI M-component (multilevel)
  - Based on GEMSOS
- Evaluation on GTNP
- This paper, however, about GEMSOS TCB only

GEMSOS A1 Formal Verification Process

- FSPM, FTLS written in InaJo specification language
- BLP BST proven using FDM theorem prover
  - FSPM was not “pure” BLP, but the GEMSOS interpretation of BLP
- Conformance of FTLS to model also proven
- FTLS also used for code correspondence and covert storage channel analysis

Value of Formal Verification Process

- "Provided formulative and corrective guidance to the TCB design and implementation"
- I.e., just going through the process helped prevent and fix errors in the design and implementation
- Required designers/developers to use clean designs
  - So could be more easily represented in FTLS
  - Prevents designs difficult to evaluate and understand

GEMSOS TCB Subsets

- Ring 0: Mandatory security kernel
- Ring 1: DAC layer
- Policy enforced at TCB boundary is union of subset policies

Each Subset has its own FTLS and Model

- Each subset was verified through a separate Model and FTLS
- Separate proofs, too
- TCB specification must reflect union of subset policies

Where in SDLC?

- Model and FTLS written when interface spec written
- Preliminary model proofs, FTLS proofs, and covert channel analysis performed when implementation spec and code written
- Code correspondence, covert channel measurements, and final proofs performed when code is finished

- Formal verification went on simultaneously with development

Goal of GEMSOS TCB Verification

- To provide assurance that TCB implements the stated security policy
- Through chain of formal and informal evidence
  - Statements about TCB functionality
  - Each at different levels of abstraction
    - Policy
    - Model
    - Specification
    - Source
    - TCB itself (hardware and software)
  - Plus assertions that each statement is valid wrt next more abstract level

Chain of Verification Evidence

- Notes:
  - Model-to-policy argument is informal
  - Spec to model argument is both formal and informal
  - Source to spec argument is code correspondence
  - TCB to source means HW and compiler validation
    - I.e., object code
    - Considered "beyond state of the art"

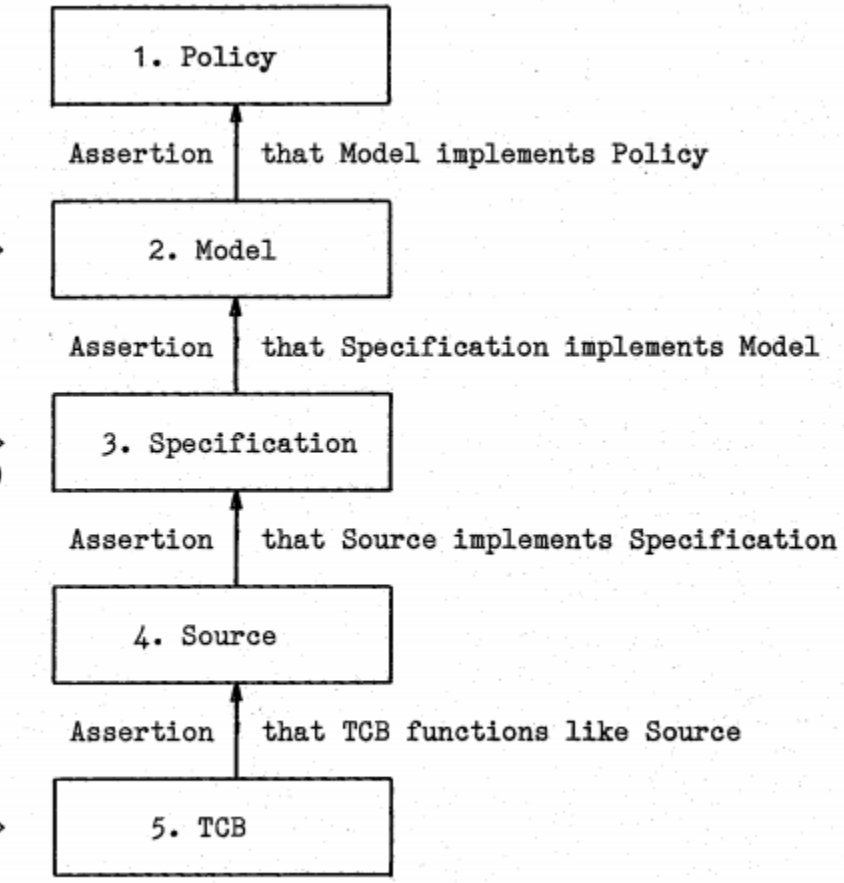


Figure 1. Chain of Verification Evidence

Model

- Mathematical statement of access control policy
- “Interpretation” of BLP
- Security defined as axioms
- Must prove all model transforms preserve axioms
  - SSC
  - *-property
  - (and probably others, as with PDP 11/45)
- Proof of model shows model upholds policy

Key Characteristic of Model

- Not just formal statement of policy or functions
- A model of a reference monitor
  - "Linchpin" of security argument
- If show that TCB satisfies reference monitor model then have shown that it is secure
  - Implies that anything outside TCB cannot violate policy
- What if did not model reference monitor?
  - May be "correct" wrt functions, but not necessarily secure

FDM “Levels”

- I.e., levels of abstraction
- InaJo language has method of formally mapping elements of one level to the elements of the next level
  - Top level: Model
  - Second level: FTLS

FTLSs

- One each for kernel and for TCB
- Exceptions, error messages, and effects visible at interface
- Transform for each call and
- Transforms for HW “read” and “write” operations
  - Other opcodes are irrelevant for access control security
- Proof maps each transform of FTLS to transform in model
- Each call specified as conditional statement
  - Last case contains any change statements
  - Exceptions specified in order of possible occurrence in code
    - Important for Covert Channel Analysis
  - Very end specifies everything else unchanged

Code Correspondence

- Three parts:
  1. Description of correspondence methodology
  2. Account of non-correlated source code
  3. Map between elements of FTLS and TCB code
- FTLS must accurately describe the TCB
- TCB must be valid interpretation of FTLS
- All security-relevant functions of TCB must be represented in FTLS
  - Prevent deliberate or accidental “trap door”

Example of Value of Formal Proof

- Subject is process/ring
- Subject can have range of access classes (trusted subject)
- Subjects in outer rings can have access class ranges "smaller" than subjects of the same process ir

```
Ring 3 Subject Range          READ WRITE
Ring 2 Subject Range      READ         WRITE
Ring 1 Subject Range  READ                 WRITE

(labels to the left dominate labels to the right)
(subject n is more privileged than subject n+1)
```

- Formal proof "stuck" trying to prove this

Example Formal Spec Detected Problem

- If range of subject in outer ring not within range of inner ring, move the outer ring access class to be within the range
- Original spec and code didn't take into account non-comparable access classes

```
dominates (ring_3_read_class, new_ring_2_read_class)
        then move (ring_3_read_class)
and
dominates (new_ring_2_write_class, ring_3_write_class)
        then move (ring_3_write_class)
```

- How to fix?

```
~dominates (new_ring_2_read_class, ring_3_read_class)
        then move (ring_3_read_class)
and
~dominates (ring_3_write_class, new_ring_2_write_class)
        then move (ring_3_write_class)
```

2nd Example Formal Spec Detected Problem

- Adjusting the access classes depends on the “move” function
- But it was found that the move function did not correctly ensure that the access class range of the outer ring subject was correct (i.e., that the “read” class dominated the “write” class)

Example of Value of Code Correspondence

- Code correspondence of kernel to spec found flaws in code:
  1. Access to segments in new child processes being checked using parent's privileges, not child's
  2. Segment descriptor in Local Descriptor Table not being set until segment brought into RAM
     - Not clear if this just meant inconsistent with model or was a real security problem

Example of Value of Covert Channel Analysis

- Two unexpected covert storage channels discovered
- Both related to “dismount_volume” call
- Dismount_volume used to (temporarily) remove set of segments from the segment structure
- Originally, any process whose access class range spanned range of volume could dismount the volume
- What if volume has only Unclassified segments?
  - TS process has made_known some of those segments
  - Unclassified process tries to dismount the volume, but gets an error message
- Fix?
  - Require caller’s range from volume low to sys-high

2nd Covert Channel

- Order of error checking
- Errors about volume could be reported to the calling subject even if subject did not have access to dismount the volume
- Fix: check label range before returning errors related to volume attributes