



INF523: Computer System Assurance

Secure Operation Continued

Prof. Clifford Neuman

Lecture 8

16 October 2020

Mid-Term Exam is Friday October 23rd



Exam will be 100 minutes, from 1PM to 2:40PM.

- A lecture will follow from 3PM to 4:20PM
- Format of the exam
 - The exam is open book, open note and online
 - 2016 and 2017 exams are posted on the class website <http://ccss.usc.edu/523>
 - The 2018 exams will be posted soon
 - Material to be covered will be the start of the semester through Secure Operation
 - You are responsible for material discussed in lecture, and also all assigned readings.

System Audit and Integrity Monitoring

- Static audit: scan systems and note discrepancies
 - Missing patches
 - Mis-configurations
 - Changed, added, or deleted system files
 - Changed, added or deleted applications
 - Added or deleted systems!
- Dynamic system integrity checking
 - Same as static, but continuous
- Example: Tripwire (<http://www.tripwire.com/>)



Tripwire

- Used to create checksums of
 - user data,
 - executable programs,
 - configuration data,
 - authorization data, and
 - operating system files
- Saves database
- Periodically calculates new checksums
- Compares to database to detect unauthorized or unexpected changes



Continuous Monitoring

- Static audit is good, but systems may be out of compliance almost immediately
- Goal: Real-time detection and mediation
 - Sad reality: minutes to days to detect, maybe years to resolve
- Need to automate monitoring
- See, e.g.,
 - SANS Whitepaper:
<http://www.sans.org/reading-room/whitepapers/analyst/continuous-monitoring-is-needed-35030>
 - NIST 800-137 Information Security Continuous Monitoring (ISCM) for Federal Information Systems and Organizations
<http://csrc.nist.gov/publications/nistpubs/800-137/SP800-137-Final.pdf>

Inventory of Systems and Software



- IT operations in constant state of flux
 - New services, legacy hardware and software, failure to follow procedures and document changes
- Make a list of authorized systems, software, and versions (and patches)
 - Create baseline
 - Discovery using administrative efforts, active and passive technical efforts
- Regularly scheduled scans to look for deviations
 - Continuously update as new approved items added or items deleted



Other things to Monitor

- System configurations
- Network traffic
- Logs
- Vulnerabilities
- Users

- To manage workload:
 - Determine key assets
 - Prioritize alerts

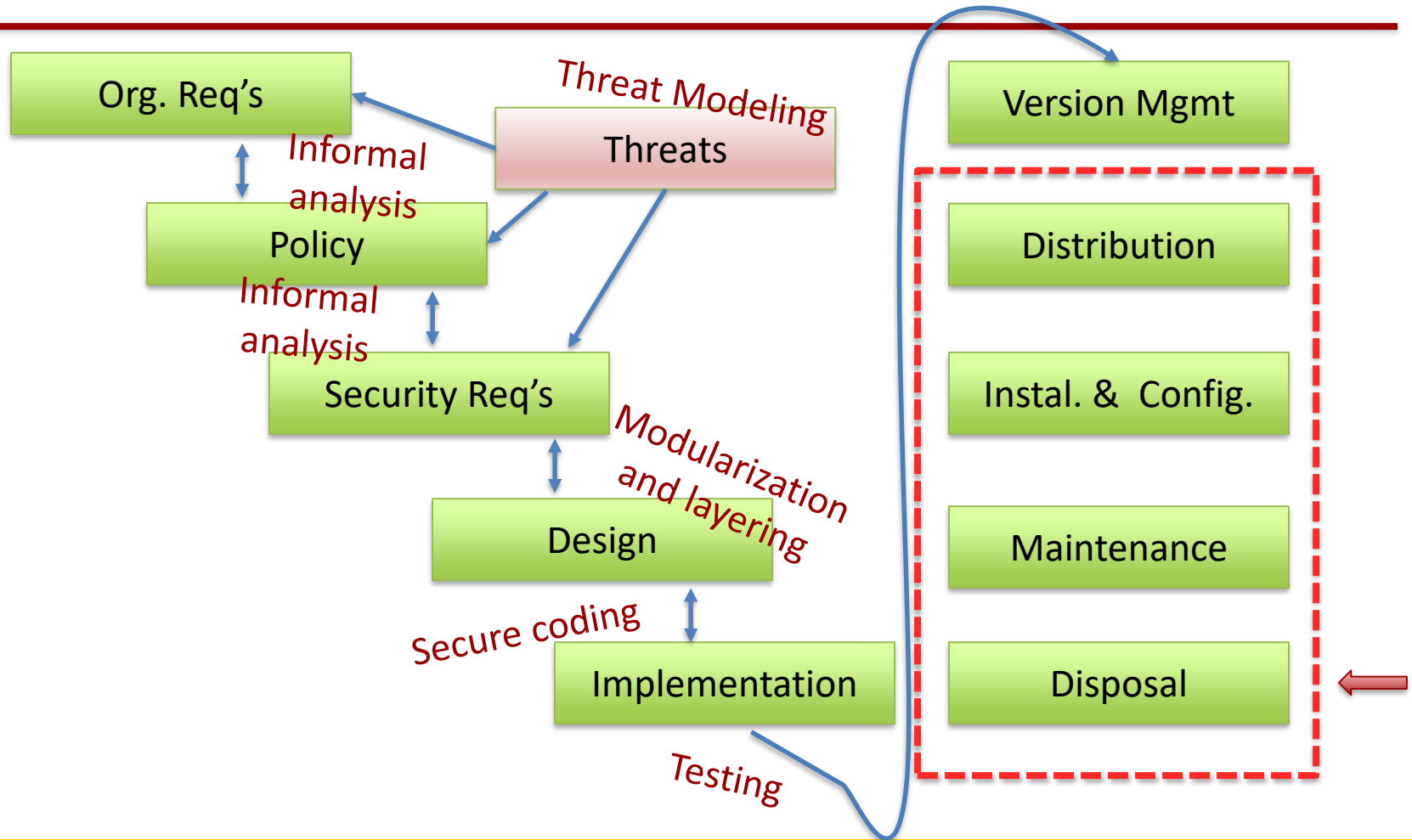
System Integrity in High Assurance System



- E.g., GTNP FER (p. 121-123)
 - HW and SW integrity tests at boot time
 - Continuously running diagnostic tests (system idle)
- Obviously standalone
- Not part of a larger networked environment



“Assurance Waterfall”



Secure Disposal Requires Attention



- Delete sensitive data on systems before disposal
 - Not always obvious where media is
- E.g., copy machines have hard drives
<http://www.cbsnews.com/news/digital-photocopiers-loaded-with-secrets/>
- E.g., mobile phones not properly erased
<http://www.theguardian.com/technology/2010/oct/12/mobile-phones-personal-data>
 - *50% of second-hand mobile phones contain personal data*



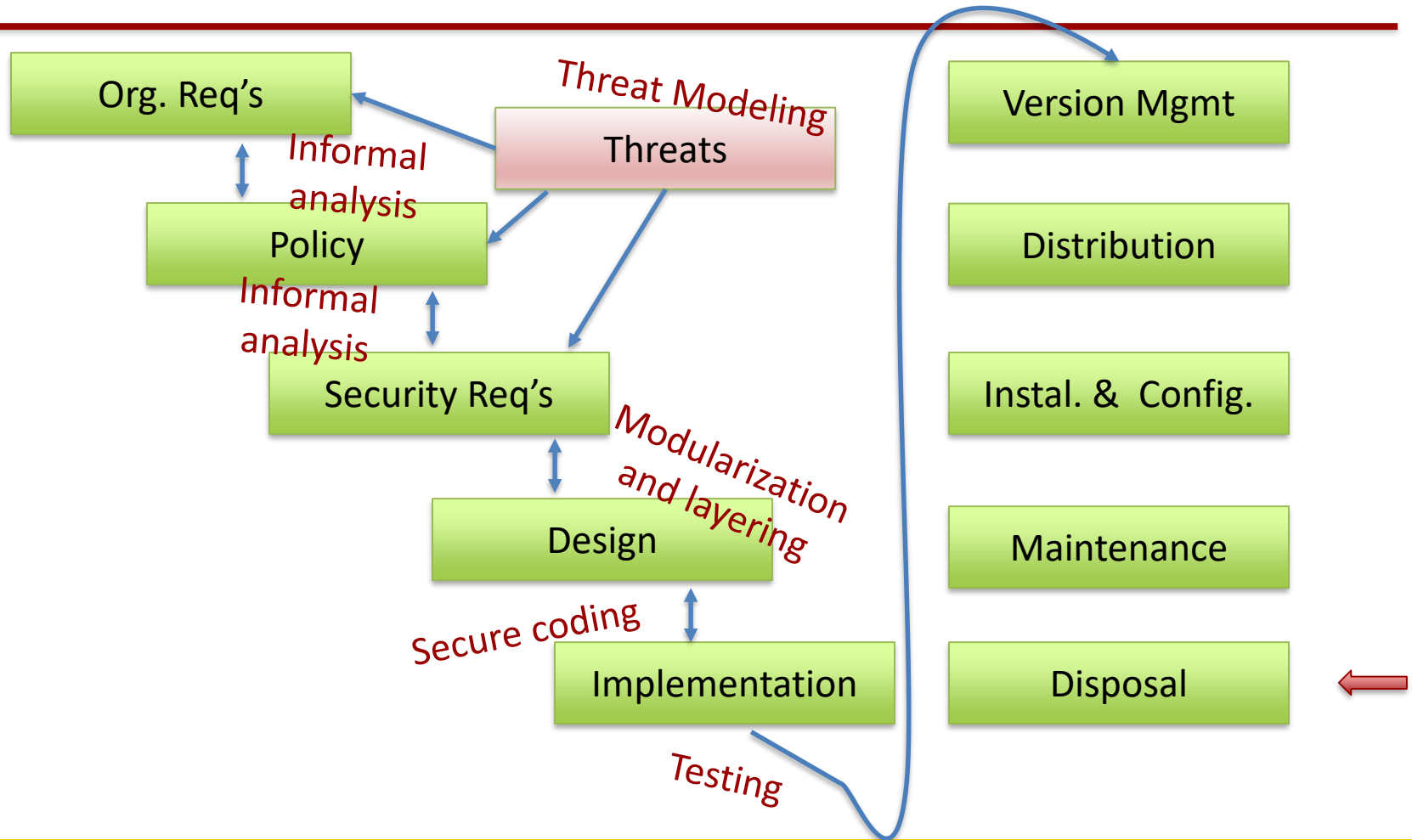
Secure Disposal

- User proper disposal techniques
 - E.g., shred drives or other storage media for best results
 - Degaussing of magnetic media not enough
 - SSDs even harder to erase





Outline of Pre-Mid-Term Course Material



Mid-Term Exam is Friday October 23rd



Exam will be 100 minutes, from 1PM to 2:40PM.

- A lecture will follow from 3PM to 4:20PM

- Format of the exam
 - The exam is open book, open note and online
 - 2016 and 2017 exams are posted on the class website
<http://ccss.usc.edu/523>
 - The 2018 exams will be posted soon
 - Material to be covered will be the start of the semester through Secure Operation
 - You are responsible for material discussed in lecture, and also all assigned readings.

'19 Mid-term Exam: Q1 - Testing



-
- a) Why might static testing based on automated source code analysis using “lint-like” tools be ineffective against an intentionally inserted vulnerabilities in a system (i.e. a subversion). Note that there is more than one reason. (5 points)

 - b) Which test techniques or tools (as discussed in our lecture on testing) are more effective at discovering previously unknown vulnerabilities? For each technique you list, explain why it might identify unknown vulnerabilities. (5 points)

 - c) For each of the following test techniques, explain why it might be more effectively applied when the tester has knowledge of the implementation of the system being tested (e.g. in a white-box test):
 - I. Fuzzing (5 points)
 - II. Penetration-testing / red-teaming (5 points)

'19 Mid-term Exam: Q2 – Software Authenticity



Please recall our discussion of Trusted Computing from Computer Science 530. In our discussion from 530 we covered how the hardware based **trusted platform module (TPM)** provides **attestation**. Specifically, the TPM maintains a running checksum (called a **measurement**) of the full software stack from the BIOS through the hypervisor (if present), the operating system, and the application, and it stores this value in a **platform configuration register (PCR)**. The TPM can then digitally sign the value of the PCR and that signed value can be used by the application in protocol exchanges with remote entities. Additionally, the TPM can compare the value of the PCR with acceptable values stored with other encryption keys to decide if those keys can be unlocked for use by an application, or applied to decrypt other data within the TPM itself.

Next, consider our discussion of secure distribution and secure update from this semester. Specifically, consider the use of digital signatures in the proper way to manage distribution and updates from that discussion (it should be in your notes). Consider also tools like tripwire and AFIK that also use checksums (but not necessarily digital signatures) to determine whether software has changed after installation or legitimate updates.

In this question you will explain the differences in the assurances provided by (i) Trusted computing and the TPM, (ii) Digital Signatures used in software and update distribution, and (iii) tools that detect changes like Tripwire and AFIK? In providing your answer, be sure to tell me:

'19 Mid-term Exam: Q2 – The Questions



- a) **What entity provides the assurance (e.g. the attestation or statement)** and how the placement of the entity affects our confidence in the findings. In answering, consider how resistant the entity providing the assurance statement is to subversion itself.
- b) **Also tell me to whom such assurance statements are provided** (i.e. who is protected, e.g. a remote system, the local administrator, or is the assurance that condition on access to data will be followed, e.g. the assurance is provided to the owner of data).
- c) Also tell me differences or similarities in the kinds of attacks/changes that might be detected (e.g. virus, worm, trojan horse, changes installed by an attacker), and more specifically **the timeline of when changes are made by an attacker that might result in detection**, and which might go undetected.

'19 Mid-term Exam: Q3 – Scenario



- **Implantable Medical Devices** – You have been hired by a medical device manufacturer to assess the security of a series of **medical devices** to be **implanted in patients** to monitor and control certain aspects of their heart. Fortunately, you have joined at the early stages of their design and you are also expected to provide concrete suggestions to improve the assurance possible for their control architecture.
- The system that is under design collects and stores data about the operation of the patient's heart, **and allows that data to be read by physicians**. The system will also monitor the data in real time and based on the analysis of that data it may deliver electrical impulses to interrupt and correct dangerous conditions that might arise. Unfortunately, the rules that are to be followed for the real time analysis and the corrective actions are not known at the time the device is implanted, and therefore **certain aspects of the programming must be updated after the device is implanted**. This process **requires communication** with **other computer systems managed by the physician** and the **device manufacturer**.
- Given the description of the system, and the parts of the system described earlier,

'19 Mid-term Exam: Q3 – Questions



-
- a) Create a data flow diagram (DFD) for the system. Remember that trust boundaries are shown in the DFD, and you can make recommendations about the placement of those boundaries (advising on the design of the device) by your placement of those lines (or the placement of the other components with respect to those lines), so try to place them in places that make sense from a security perspective, but that also make sense from a practical implementation perspective. (10 Points)
- b) Create an attack tree for the system. What might be possible end goals of an attacker (hint: it might not always be to kill the patient)? Make sure your tree shows how such goals might be achieved. Start the attack tree with your general understanding of possible goals and capabilities of an adversary, but you may come back to add to this attack tree after (or during) your completion of question 3c. (10 points)

'19 Mid-term Exam: Q3 – Questions



-
- c) List the potential weaknesses in the system according to the STRIDE model as applied to the DFD created in 3a. (10 points)
- d) Defending the system – List some of the techniques and defensive measure you could apply that will result in improved arguments for assurance in the system. Please be sure to include among these techniques those that are 1) architectural – relating to the interaction of components of the system (and the conditions under which some of those interactions may occur), 2) software based – relating to the structure of software and the programming of the components, and 3) technological – specific technological techniques or components that can be applied to improve the security and assurance of the system. (15 points) .



INF523: Computer System Assurance

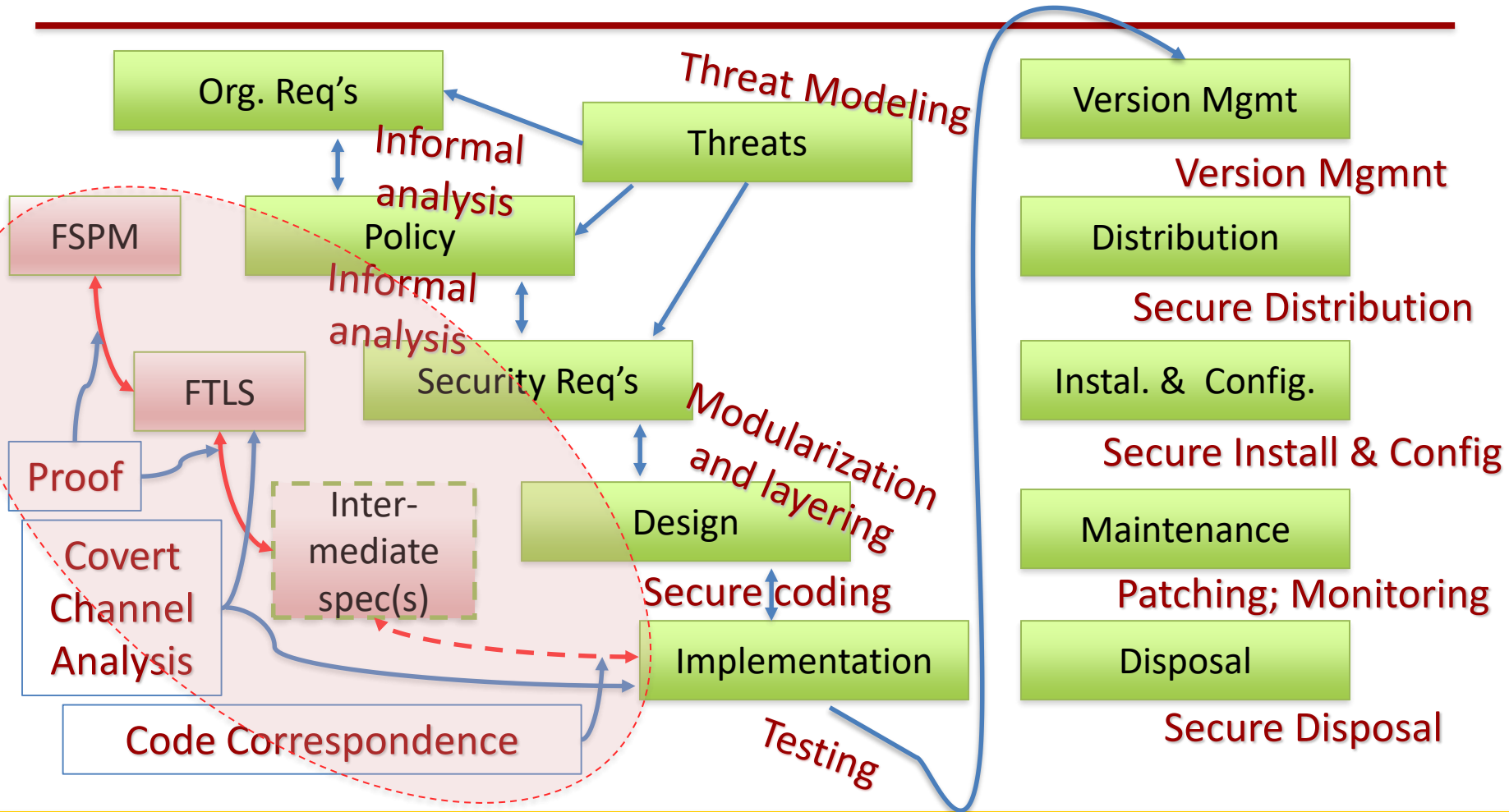
Formal Methods

Prof. Clifford Neuman

Lecture 9
23 October 2020



“Assurance Waterfall”



Reading for (this) Time (but not on mid-term)



- 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 (but not on mid-term)



- 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

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



- Formal

- Mathematical
- E.g., $\forall t. \forall x. (t \geq x \wedge (\text{sys_init}(x))) \rightarrow x(t) < 5$
- Easily handles abstractions, concise, non-ambiguous, precise, complete, etc.
- But, requires lots of training and experience to do right

Formal vs. “Informal” Verification



- “Informal” verification:
 - **Testing** of various sorts
 - Finite, can never 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 $\text{dom}(R_i)$.)

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

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

The rule:

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

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  $R_k \notin \text{dom}(R1)$  then  $R1(R_k, v) = (? , v)$ ; else if  $r \in M_{ij}$  and  $\langle [S_i \in S' \ \text{and} \ *1(R_k, v)] \ \text{or} \ [S_i \in S_T \ \text{and} \ f_s(S_i) \times f_o(O_j)] \rangle$ 
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

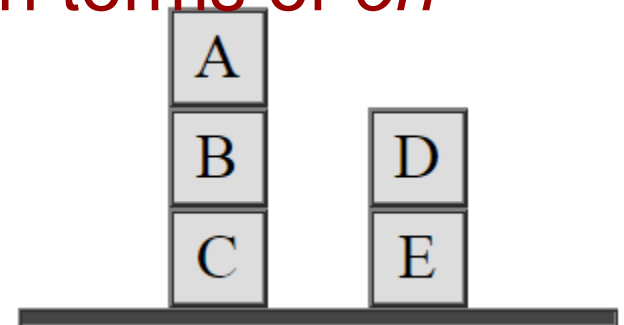


- Represents interface of the system
 - In terms of exceptions, error messages, and effects
 - Must be shown to accurately reflect TCB interface
 - Include HW/FW operations, if affect state at interface
 - TCB “instruction set” consists of HW instructions accessible at interface and TCB calls
- Describe 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



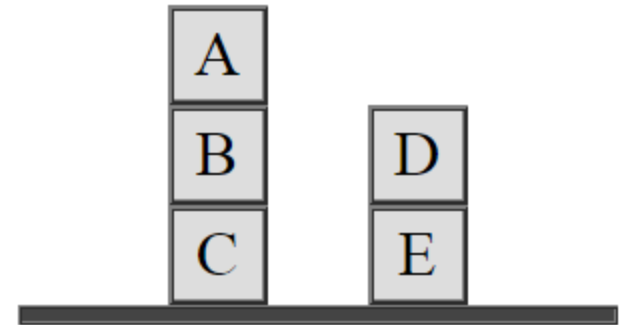
- Example, “blocks world”
- 5 objects $\{a,b,c,d,e\}$
 - Table is not an object in this example
- Relations $\{on,above,stack,clear,ontable\}$
- $on(a,b); on(b,c); on(d,e)$
- $\neg on(a,a), \neg on(b,a), \dots$, etc.
- Define all of the other relations in terms of on





Creating a Formal Specification

- Define all of the other relations in terms of *on*
- $\forall y.(\text{clear}(y) \Leftrightarrow \neg \exists x. \text{on}(x,y))$
- $\forall x.(\text{ontable}(x) \Leftrightarrow \neg \exists y. \text{on}(x,y))$
- $\forall x. \forall y. \forall z. (\text{stack}(x,y,z) \Leftrightarrow \text{on}(x,y) \wedge \text{on}(y,z))$
- $\forall x. \forall z. (\text{above}(x,z) \Leftrightarrow \text{on}(x,z) \vee \exists y. (\text{on}(x,y) \wedge \text{above}(y,z)))$



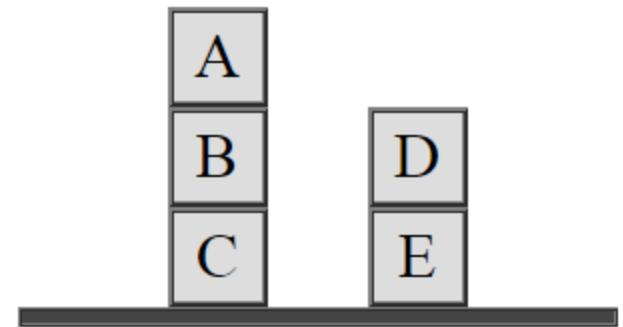
– We are missing something for *above*. What is it?

$$\forall x. \neg \text{above}(x,x)$$



Alternative Specification

- Define all of the other relations in terms of *above*
- $\forall x.(\text{ontable}(x) \Leftrightarrow \neg \exists y.\text{above}(x,y))$
- $\forall x.(\text{clear}(x) \Leftrightarrow \neg \exists y.\text{above}(y,x))$
- $\forall x.\forall y.(\text{on}(x,y) \Leftrightarrow \text{above}(x,y) \wedge \neg \exists z.(\text{above}(x,z) \wedge \text{above}(z,y)))$
- What about *stack*?
 - Can define in terms of on, as before
- Need other axioms about *above*:
- $\forall x.\neg \text{above}(x,x)$
- $\forall x.\forall y.\forall z. \text{above}(x,y) \wedge \text{above}(y,z) \Rightarrow \text{above}(x,z)$
- $\forall x.\forall y.\forall z. \text{above}(x,y) \wedge \text{above}(x,z) \Rightarrow y=z \vee \text{above}(y,z) \vee \text{above}(z,y)$
- $\forall x.\forall y.\forall z. \text{above}(y,x) \wedge \text{above}(z,x) \Rightarrow y=z \vee \text{above}(y,z) \vee \text{above}(z,y)$





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., $\forall x. \text{remove}(\text{add}(x, \text{EMPTY})) = x$
- 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

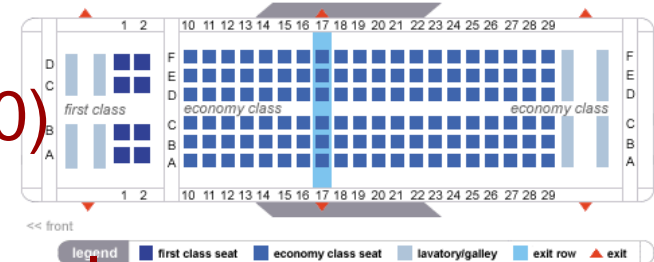


- Previous example used first-order logic (FOL)
 - \forall and \exists
- For complex systems, FOL may not be enough
- Want “higher-order” logic (HOL), which can take functions as arguments
- E.g., [Rushby PVS phone book example]
 - <http://www.csl.sri.com/papers/wift-tutorial/slides.pdf>



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 $[N \rightarrow S]$
 - 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) [$N \rightarrow S$]
 - $e0$: 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:
 - $\text{assign_seat} : [A \times N \times S \rightarrow A]$
 - $\text{deassign_seat} : [A \times S \rightarrow A]$
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



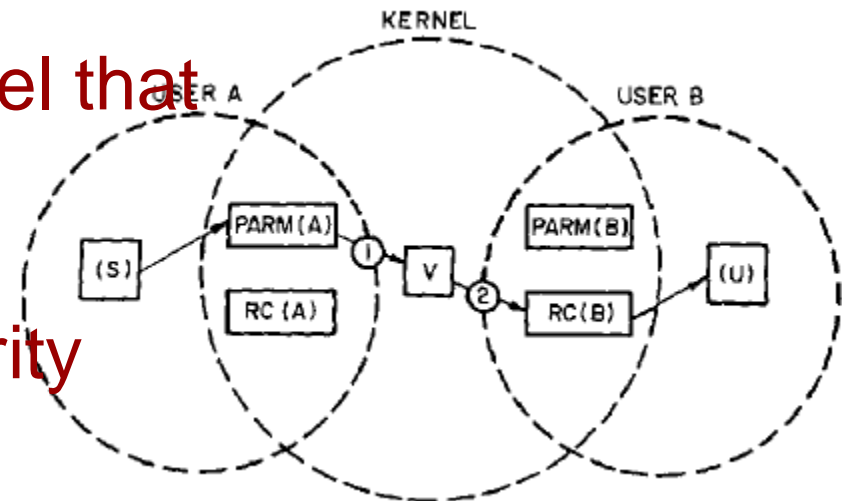
V- and O-functions

- 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., $\neg(\text{PS_SEG_INUSE}(\text{TCP}, \text{dseg})) \wedge \text{RC}(\text{TCP}) = \text{NO}$
 - 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
   [not AST_WAL(dpn,TCP) or
    AST_TYPE(dpn,TCP) ≠ DIRECTORY or
    DIR_SIZE'(dpn,entry) = 0]
B  then RC(TCP) = NO
C  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)
                  = acle + 1 (mod acle_max + 1)
F(pn)   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 $(f \wedge g) \vee (\neg f \wedge h)$

```
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
    not AST_WAL(dpn,TCP) or
B  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
D  if 'DIR_ACL_HEAD'(dpn,entry) ≠ 0
    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)
                = acle + 1 (mod acle_max + 1)
        else VM(pn) = 0
        end and
        WAIT_SET(pn) = ∅ and
        SMFR_COUNT(pn) = 1
        and (∀aproc)
F(pn)

```

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



```
A   if not PS_SEG INUSE(TCP,dseg) or  
    not AST_WAL(dpn,TCP) or  
B   AST_TYPE(dpn,TCP) ≠ DIRECTORY or  
    'DIR_SIZE'(dpn,entry) = 0  
C   then RC(TCP) = NO
```

- 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



```
A   if not PS_SEG_INUSE(TCP,dseg) or  
    not AST_WAL(dpn,TCP) or  
B   AST_TYPE(dpn,TCP) ≠ DIRECTORY or  
    "DIR_SIZE"(dpn,entry) = 0  
C   then RC(TCP) = NO
```

- *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

A	if <code>not PS_SEG_INUSE(TCP,dseg)</code> or	Subject/ Case	Condi- tions	Ef- fects	Read Levels	Write Levels
B	<code>not AST_WAL(dpn,TCP)</code> or					
	<code>AST_TYPE(dpn,TCP) ≠ DIRECTORY</code> or					
	<code>'DIR_SIZE'(dpn,entry) = 0</code>	S_i/I	A	C	$PL(TCP)$	$PL(TCP)$
C	then <code>RC(TCP) = NO</code>	S_i/II	$\bar{A} \ B$	C	$PL(TCP), L(dpn)$	$PL(TCP)$

- Second case: $\neg A \wedge B \wedge C$
 - $PS_SEG_INUSE(TCP,dseg) = TRUE$ 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: $L(dir) \leq L(RC)$
- I.e., $L(dpn) \leq PL(TCP)$



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 $L(pn) \leq PL(proc)$
- 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”

Proof -->

C.Channel -->
Analysis (CCA)

Testing -->
& CCA

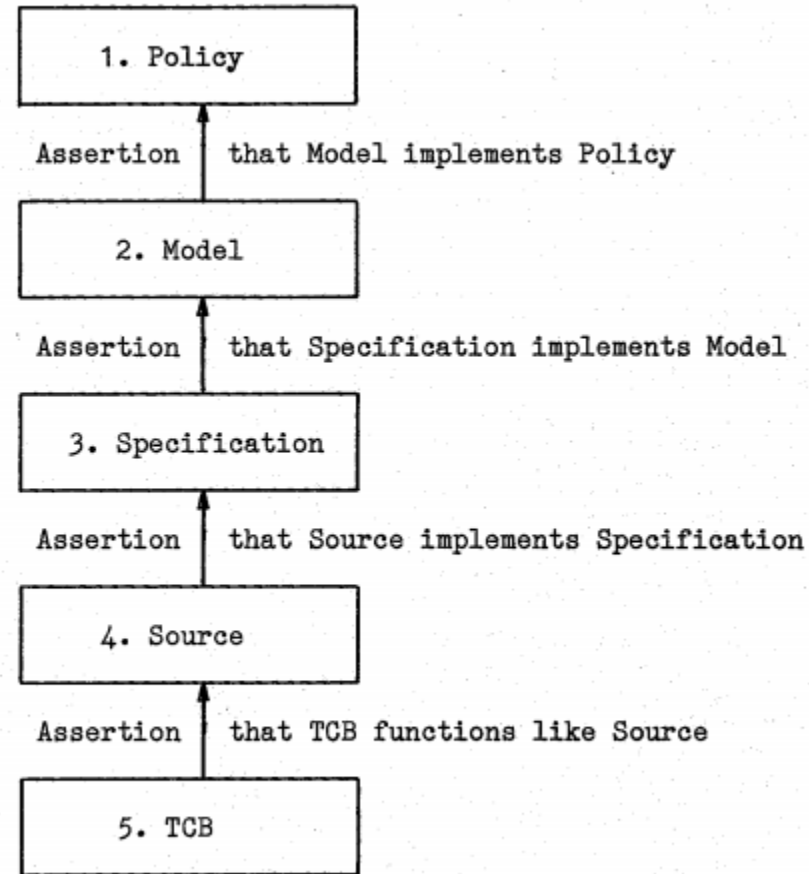


Figure 1. Chain of Verification Evidence



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