# An Analysis of the GWV Security Policy

Jim Alves-Foss and Carol Taylor
Center for Secure and Dependable Systems
University of Idaho

Work supported in part through contracts/grants from DoD and Lockheed-Martin Areo.

# Introduction

- Safety and Security
  - Need for computer systems to operate safely and securely
  - Specification and verification of non-functional system properties is not straightforward
    - How do you make systems safe? Or, secure?
  - Safety involves a system behaving in a specified way
  - Security involves a system behaving in a way that is not disallowed

# Introduction

- Certification
  - Critical systems that must operate securely or safely go through external certification
  - For safety, FAA certification process for avionics software
    - Do-178B development criteria
  - For security, US government certification process for software assurance
    - Common Criteria Evaluation Assurance Levels (EAL 1 through 7)

# Introduction

- Common Criteria (CC) Requirements
  - Developers must follow development standards that include security requirements
    - At the highest assurance levels, a formal security policy is required
      - Prove formally that a functional specification satisfies the formal security policy

# Formal Methods and Security

- Formal Methods for Security (the CC approach)
  1. There is a formal security policy; and proofs that the policy satisfies the requirements.
  2. There is a formal functional specification of the system; and proofs that it satisfies the policy.
  3. ...
- The aim of this paper is to show that there are shortcomings in the presentation of GWV that prohibits the requisite proofs of #1.

# Security Policies

- A security policy can be defined as specifying the “authorized” and “unauthorized” states of a system
  - We can say system A satisfies policy P
- A formal security policy is used to specify the “performance” or “behavior” of the system.
  - We can say policy P meets requirement R
- We use this to say A meets R

# Security Policy Misuse

- For a policy to be viable, there must be a statement of the class of systems it applies to.
- **Example:**
  - A masterlock padlock may be a strong device for limiting access.
  - A policy could say Bob can only open the padlock if Bob has a key. We can then prove many access control requirements given this policy.
  - However, a system that locks a brown paper sack with the padlock is not secure.
  - What went wrong? We did not place restrictions on the system – we did not say that the system must prevent other accesses. This is a common problem in the development of secure systems.

# Introduction

- Greves, Wilding and Vanfleet Policy
  - In 2003, Greves, Wilding and Vanfleet presented a formal security policy for a separation kernel
    - GWV's policy will be used in a CC certification of a separation kernel
  - In analyzing GWV, several ambiguities were discovered
    - Important concepts were also left out of the original paper
  - This paper is an attempt to clarify GWV
  - Concepts are introduced important to understanding the intent of the policy

# Overview

- Review of GWV Policy
- Clarification of GWV Policy
- Modifying the GWV Policy
- Conclusion

# GWV Policy

- The GWV policy models a separation kernel that supports partitioning
- ACL2 functions are introduced that capture the partitioning concept
  - ((current *) ⇒ *) returns the current executing partition given a machine state
  - ((segs *) ⇒ *) returns the list of segments associated with a partition
  - ((select * *) ⇒ *) returns a value associated with a memory segment in a specific state

# GWV Policy

- In a partitioning system there are constraints on communication between entities
- GWV models this by a function *direct interaction allowed (dia)*
  - ((dia *) $\Rightarrow$ *) is the set of segments allowed to communicate with a seg
  - ((next *) $\Rightarrow$ *) returns the next machine state representing one step of computation

# GWV Policy

- Another function, selectlist, accepts a segment list and returns a list of values associated with those segments

```
(defun selectlist (segs st)
 (if (consp segs)
    (cons
        (select (car segs) st)
            (selectlist cdr segs) st))
  nil))
```

# GWV Policy

## Policy states

For any given segment, *seg*, its values can only change as a result of interaction from memory segments in *dia* and part of executing partition, *current*

```
(let ((srcsegs (intersection-equal (dia seg) (segs (current st1)))))
  (implies
    (and
      (equal (selectlist srcsegs st1) (selectlist srcsegs st2))
      (equal (current st1) (current st2))
      (equal (select seg st1) (select seg st2)))
    (equal
      (select seg (next st1))
      (select seg (next st2))))))
```

# Clarification of GWV

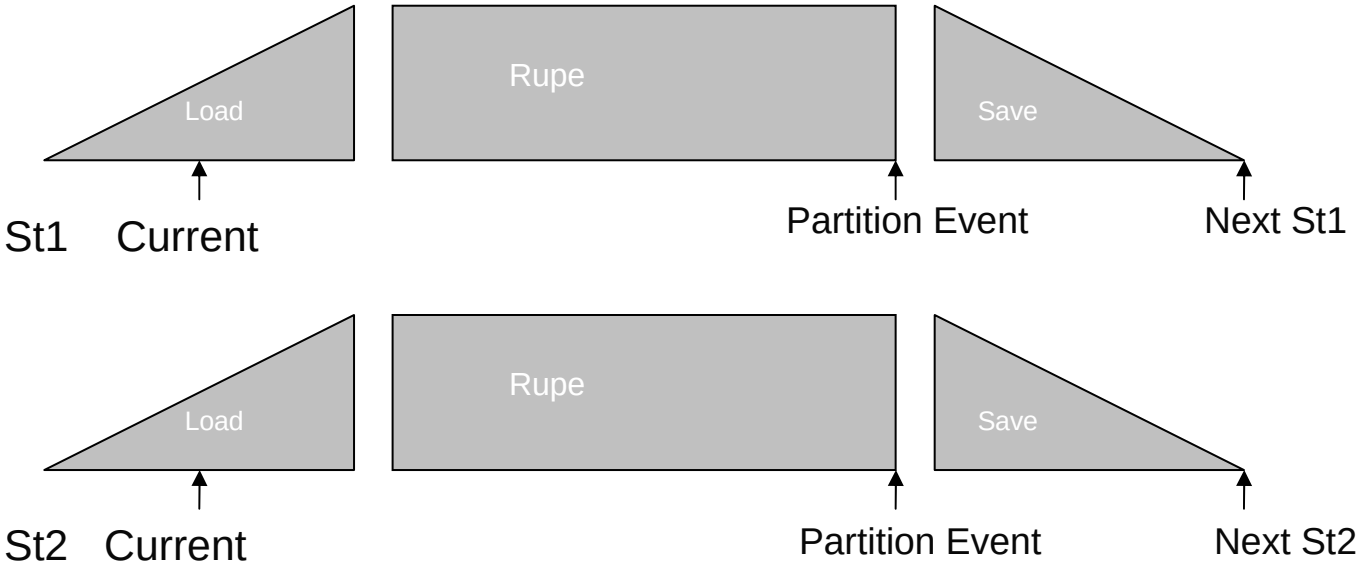
- *Next* function is one concern with GWV
  - What does this function do?
- Concept of a *cut point*
  - Point in the execution where previous partition's microprocessor state has been saved
  - Next partition has not been loaded
  - Next partition to be executed is the *current* partition

# Clarification of GWV

- Next execution involves several steps
  - Saved microprocessor state of *current* is loaded
  - *current* executes until a partition event occurs called, *run-until-partition-event (rupe)*
  - At partition event, microprocessor state saved back into memory
  - Microprocessor is sanitized of partition information

# Clarification of GWV

- How *rupe* works in two separate universes, St1 and St2

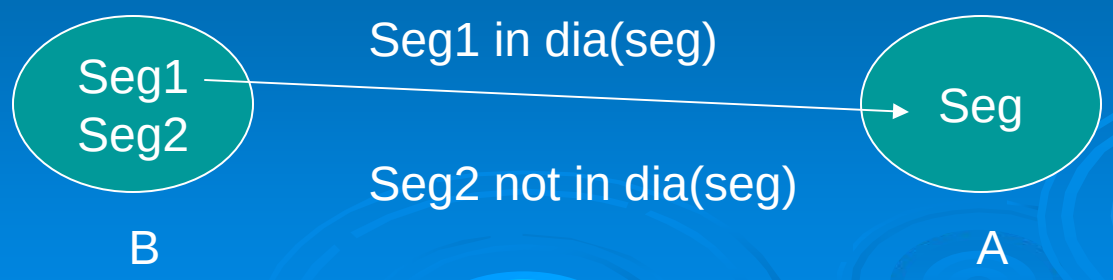


# Clarification of GWV

- Next function critique
  - No requirement that *next* be one microprocessor instruction or a set of instructions
  - In cut-point model, *next* implements many microprocessor instructions
  - Must then assume that externally visible changes to state between cut-points are not security relevant

# Clarification of GWV

- Dia is another point of concern
  - *segs* refers to memory segments of a partition including code and saved state segments
  - *dia* is the instantiation of the security policy in the separation kernel
    - Yet *dia* function as stated in GWV would allow unauthorized information flow from Seg2



# Modifying GWV

- Limiting Flow Based on Source Segments
  - To stop a copy from an unauthorized segment from copying information to a register and copying it back
  - Need to specify a restriction on the *dia* function, *dia-complete*

```
(defthm dia-complete
  (implies
    (member-equal seg (segs part))
    (subsetp-equal (segs part) (dia seg)))
```

  - Specifies that the set of segments that can influence *seg* include all segments from a given partition

# Modifying GWV

- Limit Flow Based on Code Trustworthiness
  - All state aspects of a partition must be represented by the segments
  - If some state is not mapped to a segment there could be leakage of information
    - GWV could allow a process which is not trusted to write information to a segment
    - Can happen because information flow is only specified in terms of the source of the information not who is actually doing the transferring

# Modifying GWV

- The following defthm shows the consequences of untrusted writing

```
(defthm untrusted-writing
   (implies
     (and
        (not(equal(select outbox (next st1) (select outbox (next st2)))
       (equal (current st1) current st2)))
     (equal (current st1) ‘firewall)))
```

- *untrusted-writing* shows that the contents of *outbox* could change as a result of an untrusted process

# Conclusion

- Advantage of formal models is that they communicate precisely the desired behavior of a system
  - Assumptions must be stated explicitly especially when modeling security policies
  - Security policies that will be instantiated by specific implementations must clearly state the circumstances under which the policy is both valid and invalid
- For the GWV policy, we discussed ways that a system could be insecure and still satisfy the policy
  - We suggested enhancements to GWV which we believe creates a policy that more accurately represents an abstraction of a separation kernel

# The End

- Questions?