Enforceability Theory

Language-based Security

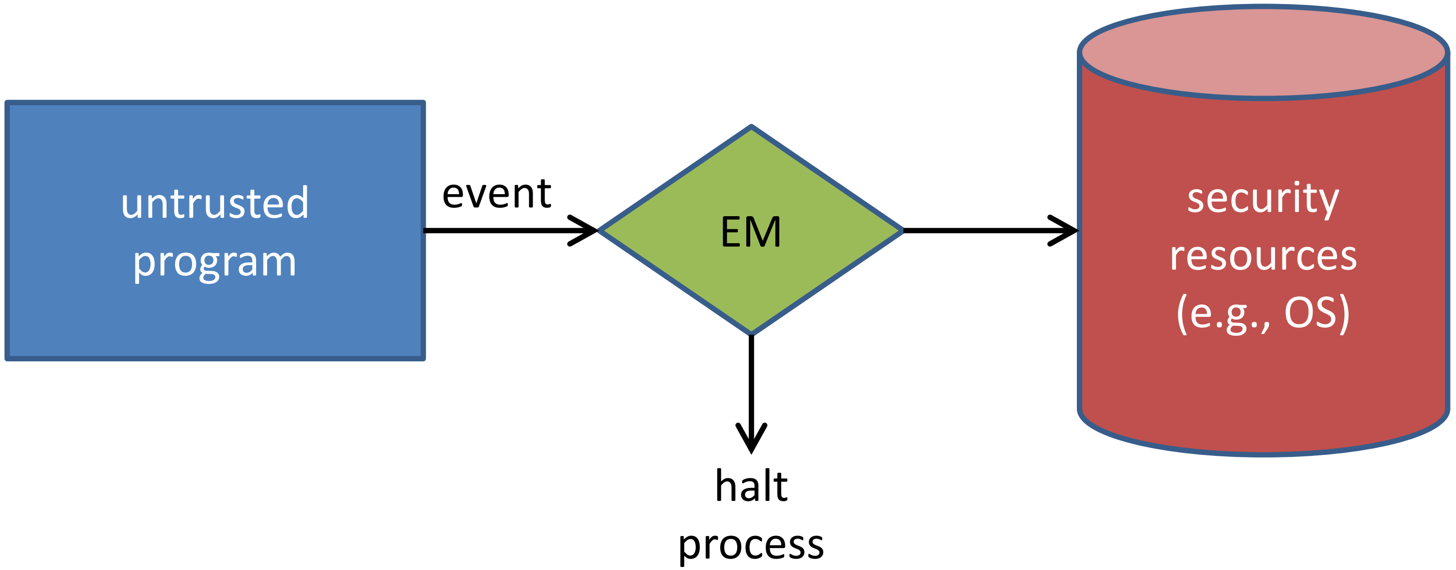
Dr. Kevin W. Hamlen

Motivating Questions

- Can we prove that mechanism M enforces policy P?
  - What is the mathematical definition of a policy?
  - What does it mean to “enforce” a policy?
- Are there limits to what is enforceable?
  - Which enforcement approaches are best suited to which policies?
  - Are there some policies that are completely beyond any known enforcement strategy?
  - Are some enforcement approaches strictly more powerful than others?
- What is the mathematical landscape of policies, policy classes, and enforcement mechanisms?

Enforceable Security Policies
[Schneider, TISSEC 2000]

- Proposed a theory of Execution (a.k.a. Reference) Monitors (EMs)
  - EMs watch untrusted programs at runtime
  - impending events mediated by the EM
  - impending violations solicit EM interventions (termination)
- Example: File system access control
  - EM is inside the OS
  - decides policy violations using access control lists (ACLs)



Programs and Policies

- An *execution* χ is a sequence of security-relevant program *events* e or *actions*
  - sequence may be finite or (countably) infinite
  - simplifying formalism: Model program termination as an infinite repetition of e_{halt}
  - now all executions are infinite length sequences
- A program Π is a SET of possible executions
  - one execution for each possible input
    - input can be an infinite sequence read over time
    - model non-determinism/randomness as an implicit input
- A policy P is a PROPERTY of programs
  - partitions the space of all programs into two groups: permissible programs and impermissible ones
  - impermissible programs are censored somehow (e.g., terminated on violating runs)

EM-enforceable Policies

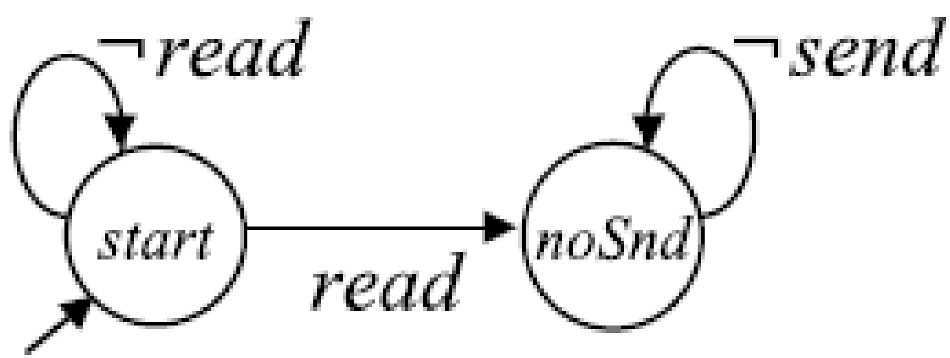
1) P(Π) ≡ $\forall \chi$□ . $\widehat{\boldsymbol{P}}(\chi)$
   - EM policies are expressible as universally quantified predicates over executions
   - P̂ sometimes called the policy's "detector"
2) Detector $\widehat{\boldsymbol{P}}$ must be prefix-closed
   - $\widehat{\boldsymbol{P}}(\chi e) \Rightarrow \widehat{\boldsymbol{P}}(\chi)$
   - $\widehat{\boldsymbol{P}}(\varepsilon)$
3) If $\widehat{\boldsymbol{P}}$ rejects something, it must do so in finite time
   - $\neg\widehat{\boldsymbol{P}}(\chi) \Rightarrow \exists\ i\ .\ \neg\widehat{\boldsymbol{P}}(\chi[..i])$

- Main discovery #1:
  - A policy satisfies (1), (2), and (3) if and only if it is a *safety policy*
  - Lamport 1977: Safety policies say that some "bad thing" never happens
  - EMs enforce safety policies!

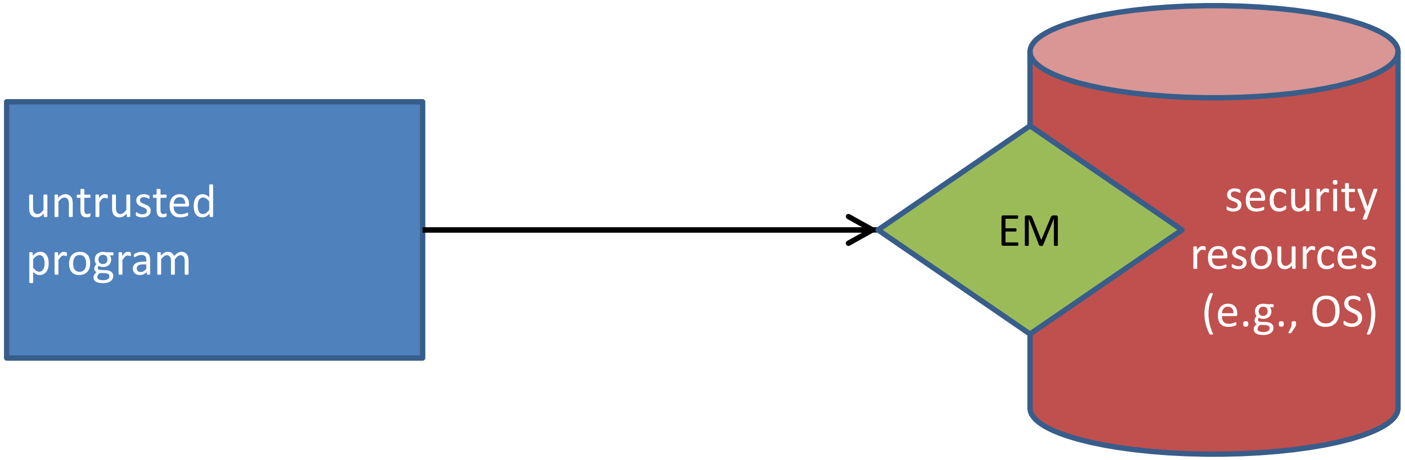
Security Automata

[Erlingsson & Schneider, NSPW '99]

- Formalization of safety policies
  - finite state automaton
  - accepts language of permissible executions
  - alphabet = set of events
  - edge labels = event predicates
  - all states accepting (language is prefix-closed)
- Example: no sends after reads

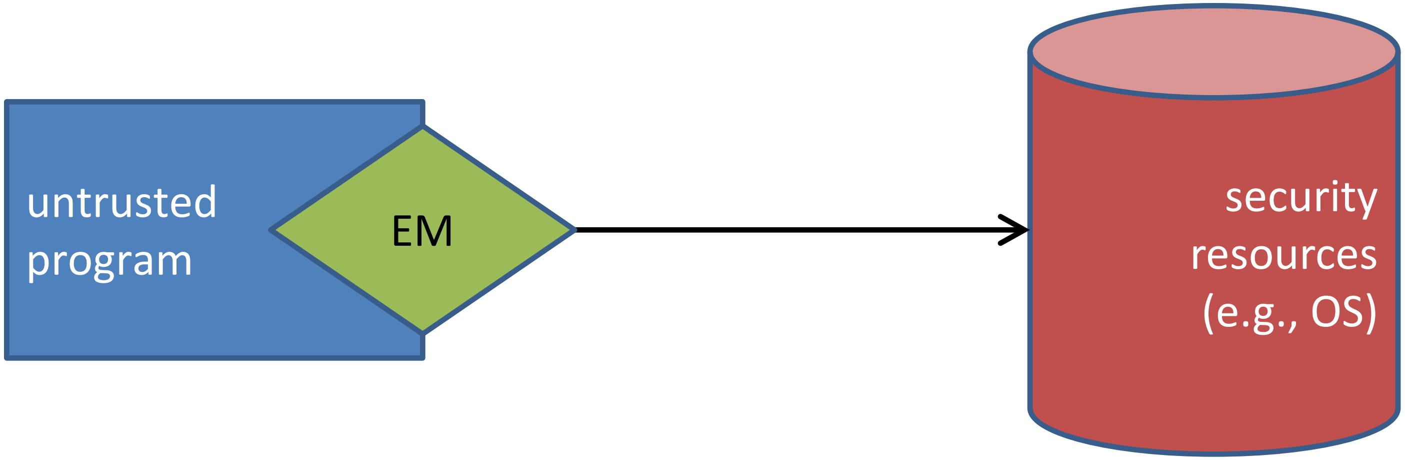


In-lined Reference Monitors



- Disadvantages of traditional EMs
  - inefficient: context-switch on every event
  - large TCB: EM extends the OS
  - weak: EM can’t easily see internal program actions
  - non-modular: changing policy requires changing OS

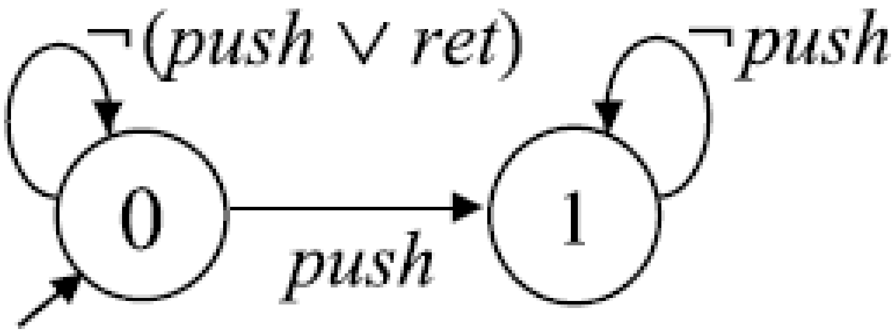
In-lined Reference Monitors



- Main idea:
  - Implement a reference monitor by *in-lining* its logic into the untrusted code
  - In-lining procedure should be automated
- Challenges:
  - How to automatically generate EM code?
  - How to preserve (non-violating) program logic?
  - How to prevent (malicious) programs from corrupting the EM?

In-lining a Security Autoamton

Example: Let's in-line this security automaton



(Policy: push exactly once before returning)

into this binary code

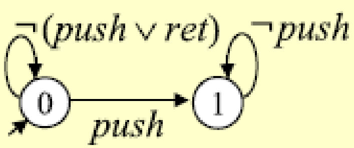
```
mul r1,r0,r0
push r1
ret
```

In-lining Algorithm

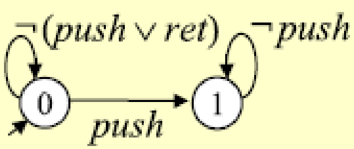
1) Conceptually in-line the automaton just before EVERY event
2) Partially evaluate (i.e., specialize) the automaton edges to the event it guards
   – some edges disappear entirely
3) Generate guard code for the remaining automaton logic

In-lining Example

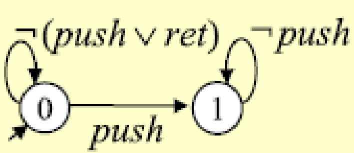
Insert security automata




```
mul r1,r0,r0
```

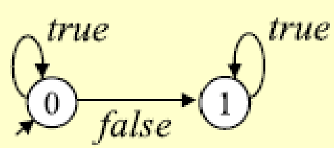



```
push r1
```

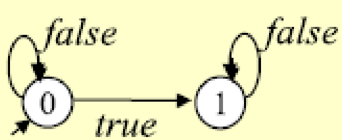



```
ret
```

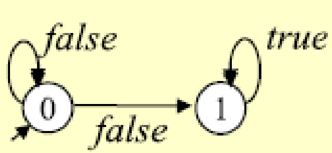
Evaluate transitions




```
mul r1,r0,r0
```

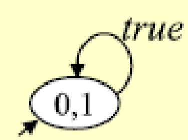



```
push r1
```



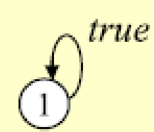

```
ret
```

Simplify automata




```
mul r1,r0,r0
```


```
push r1
```




```
ret
```

Compile automata

```
mul r1,r0,r0
 if state==0
 then state:=1
 else ABORT
push r1

 if state==0
 then ABORT
ret
```

Computability Classes For Enforcement Mechanisms

Hamlen, Morrisett, and Schneider

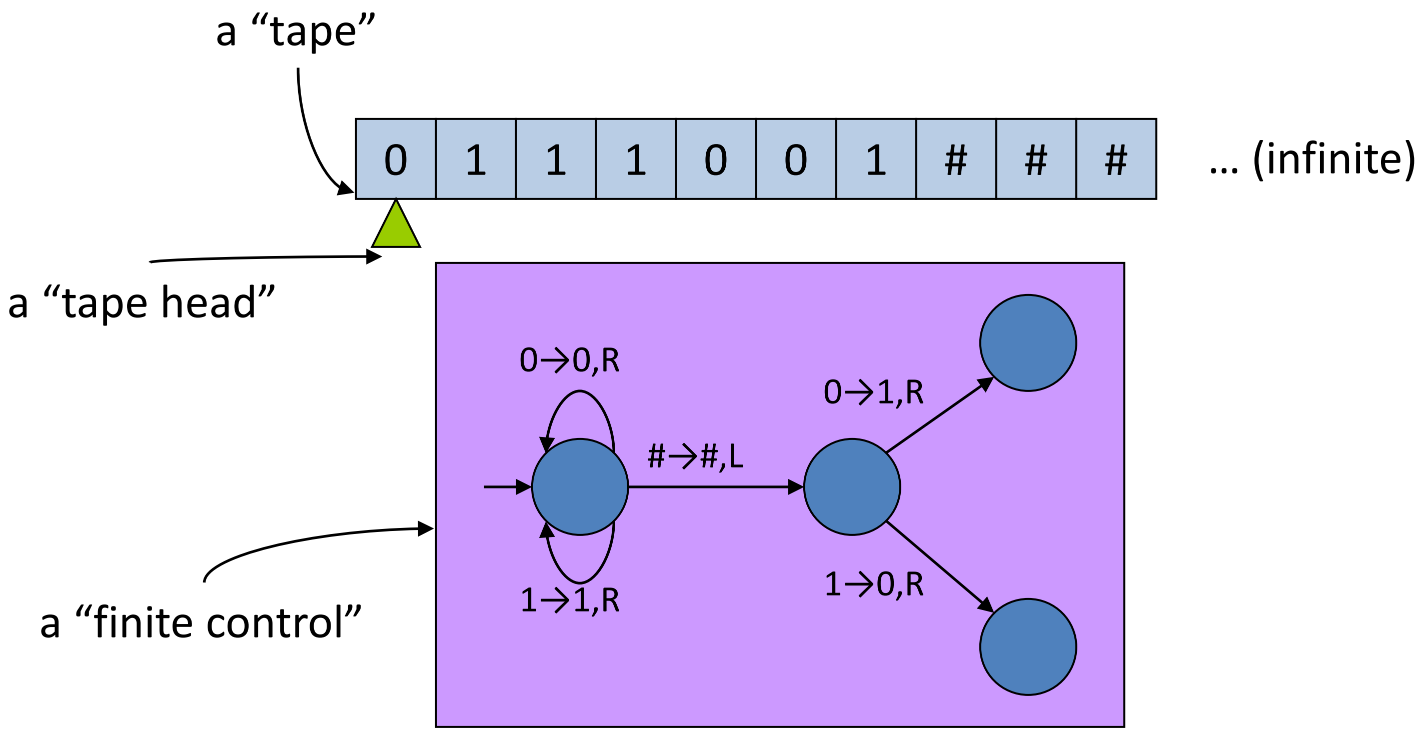
TOPLAS 2006

IRMs vs. EMs

- Implicit assumption of the Schneider paper:
  - in-lining is just an implementation strategy
  - doesn't affect set of enforceable policies
- Are we sure?
- Two interesting issues:
  - A policy constrains a program, right? But now the EM is *part* of the program. Can it constrain itself?
  - EM was previously a black box. But now it's subject to the laws of the computational model.
- Big idea: Is there a link between computability and enforceability?

Review: Computation Theory

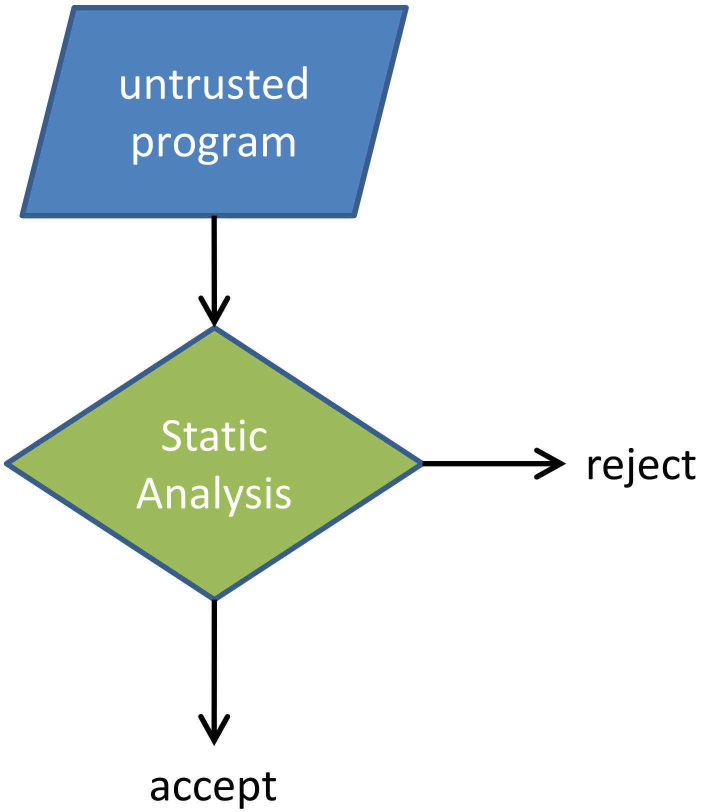
- Turing Machine
  - Alan Turing (1936)
  - simple mathematical model of a computer
  - consists of:



TM Power

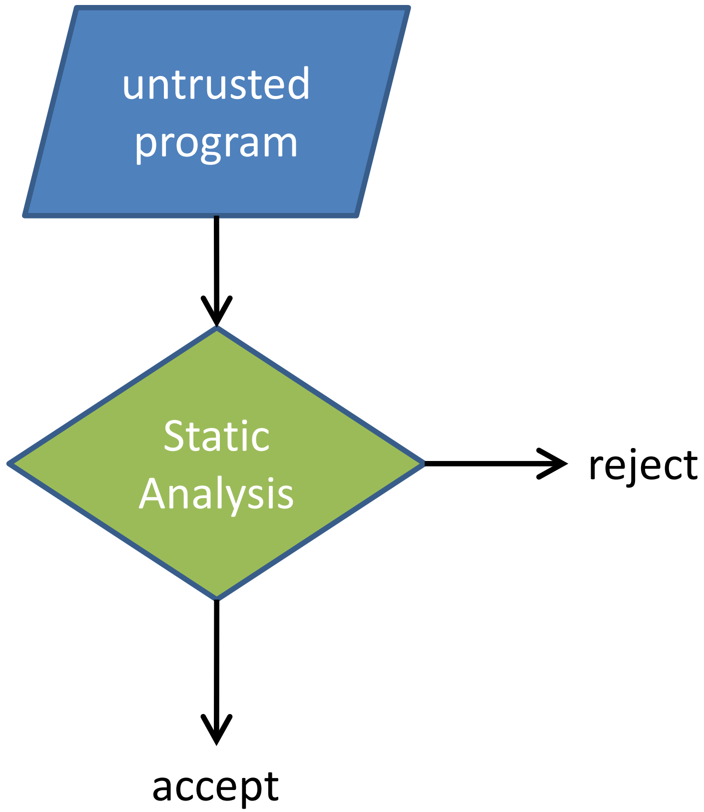
- Can do simple arithmetic
- TMs don’t necessarily terminate
- Can do anything programmable with logic gates (AND, OR, XOR, …)
- Can evaluate a C program encoded in binary
- Can simulate arbitrary TMs (given as input) on arbitrary inputs (given as input)
  - called a “universal TM”
- Intuition: Can do anything a real computer can do (but very, very slowly)
- But TMs can’t solve undecidable problems (e.g., halting problem)

Enforcement Strategy #1: Static Analysis



- Approach:
  - analyze untrusted code BEFORE it runs
  - return “accept” or “reject” in finite time
- Pros:
  - immediate answer
  - code runs at full speed
- Cons:
  - high load overhead
  - weak in power…?

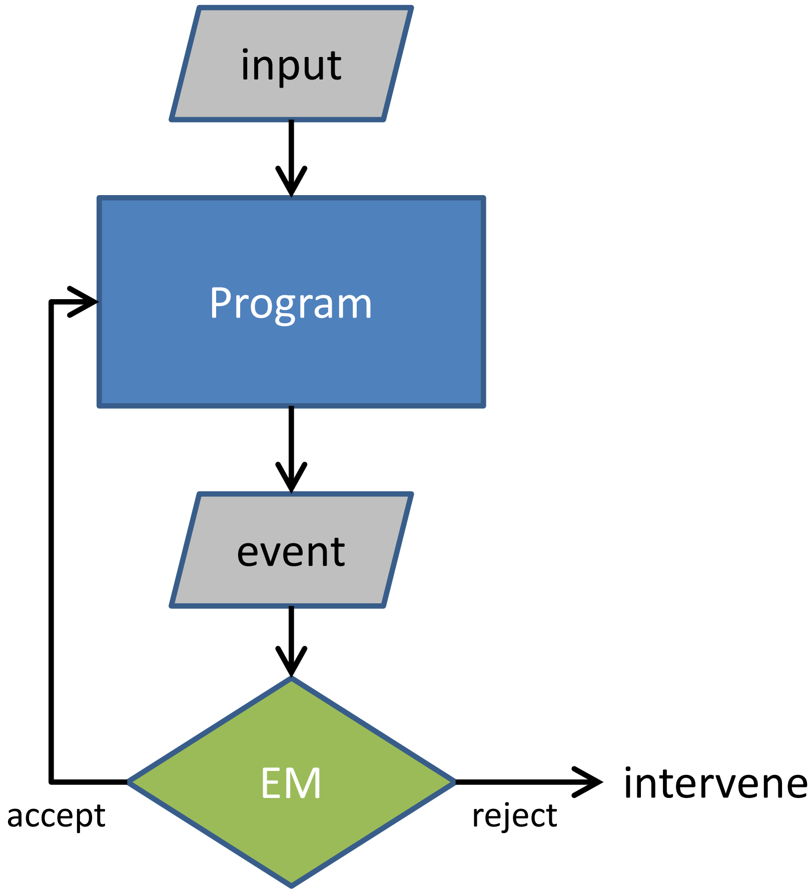
Enforcement Strategy #1: Static Analysis



- Approach:
  - analyze untrusted code BEFORE it runs
  - return “accept” or “reject” in finite time
- Pros:
  - immediate answer
  - code runs at full speed
- Cons:
  - high load overhead
  - weak in power…?

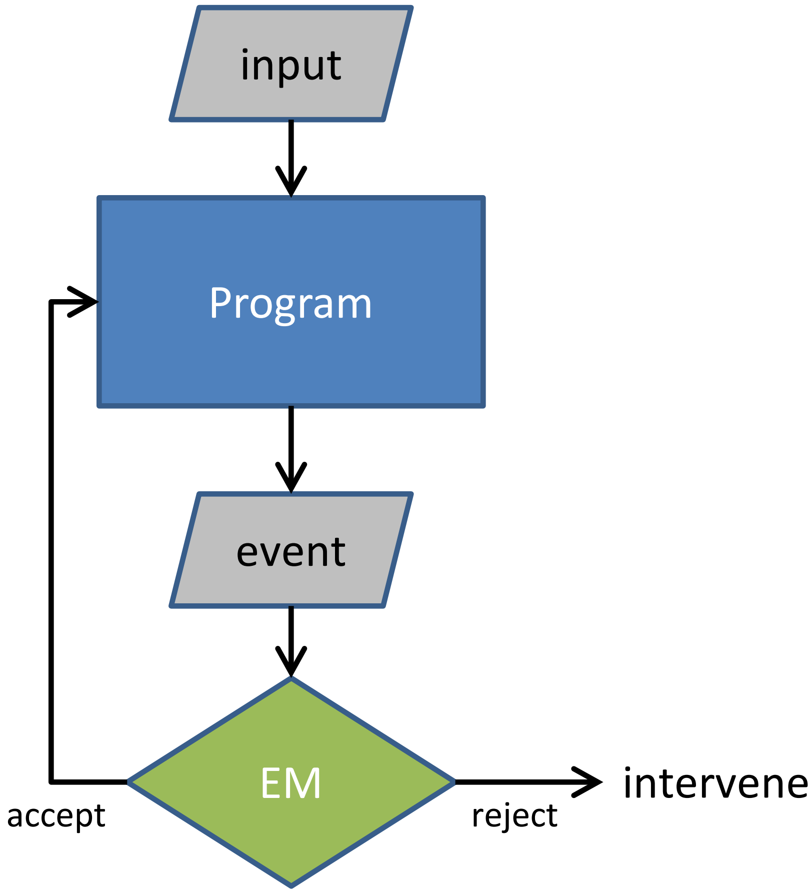
Recursively Decidable Policies

Enforcement Strategy #2: Execution Monitoring



- Approach:
  - EM monitors events
  - intervenes to prevent violations
  - implemented outside program
- Cons:
  - no answer until execution
  - runtime slow-down (context-switches)
- Pros:
  - lower load-time overhead than static analysis
  - more powerful…?

Enforcement Strategy #2: Execution Monitoring

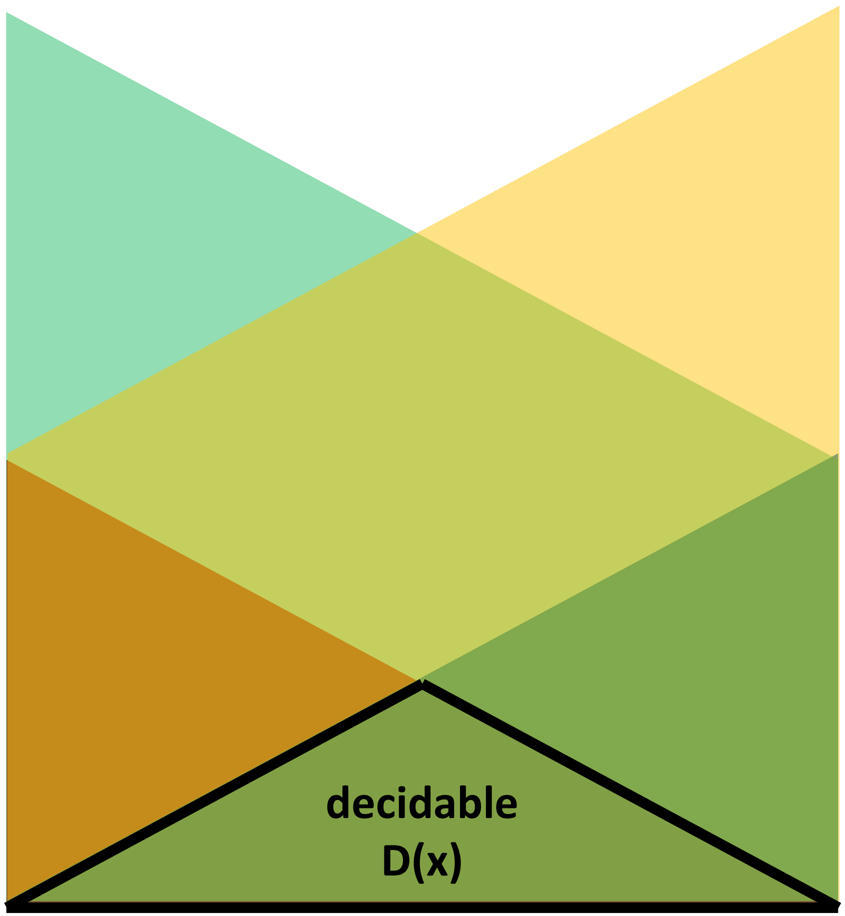


- Approach:
  - EM monitors events
  - intervenes to prevent violations
  - implemented outside program
- Cons:
  - no answer until execution
  - runtime slow-down (context-switches)
- Pros:
  - lower load-time overhead than static analysis
  - more powerful...?

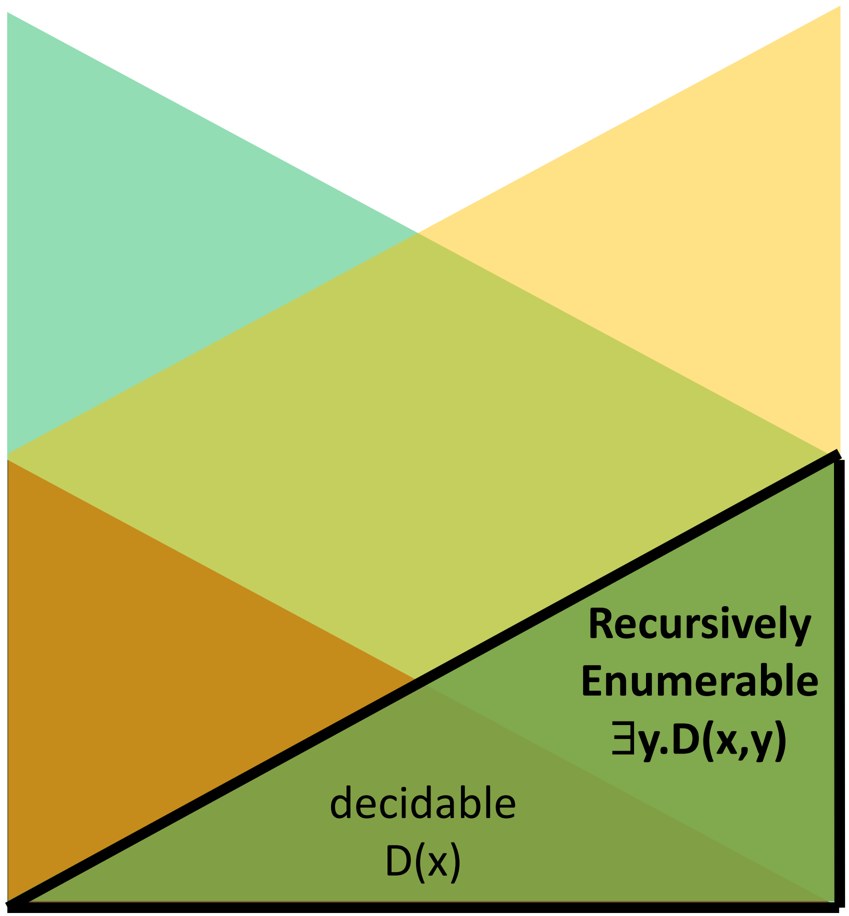
co-Recursively Enumerable Policies

Arithmetic Hierarchy

Arithmetic Hierarchy



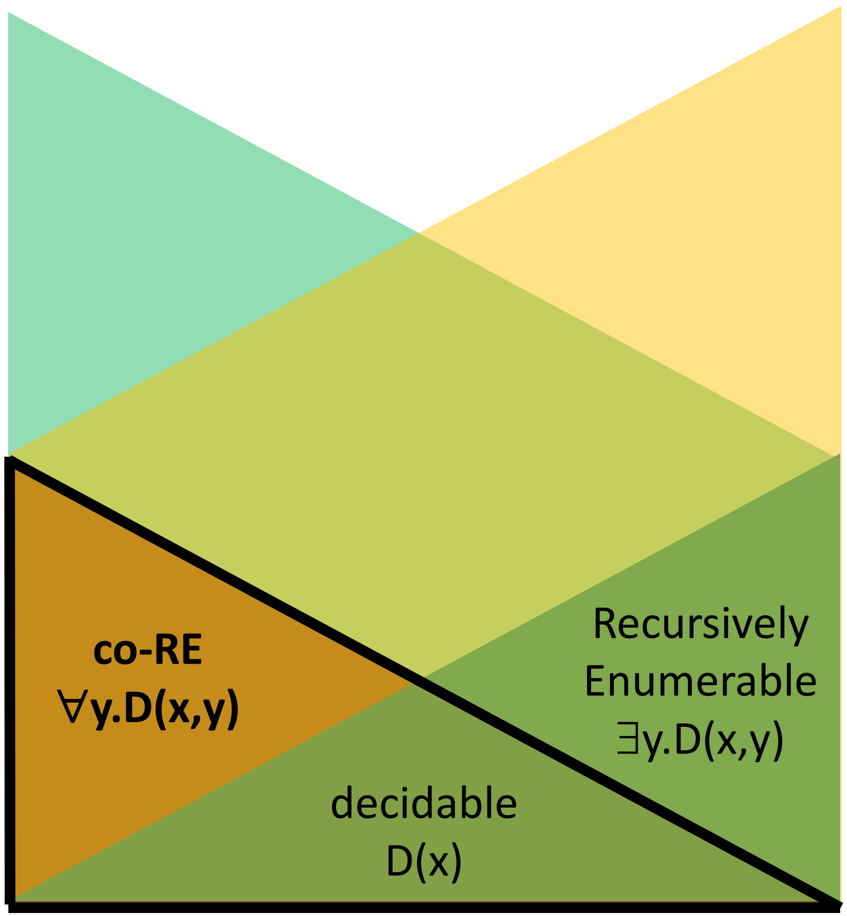
Arithmetic Hierarchy



Example: TM x eventually halts

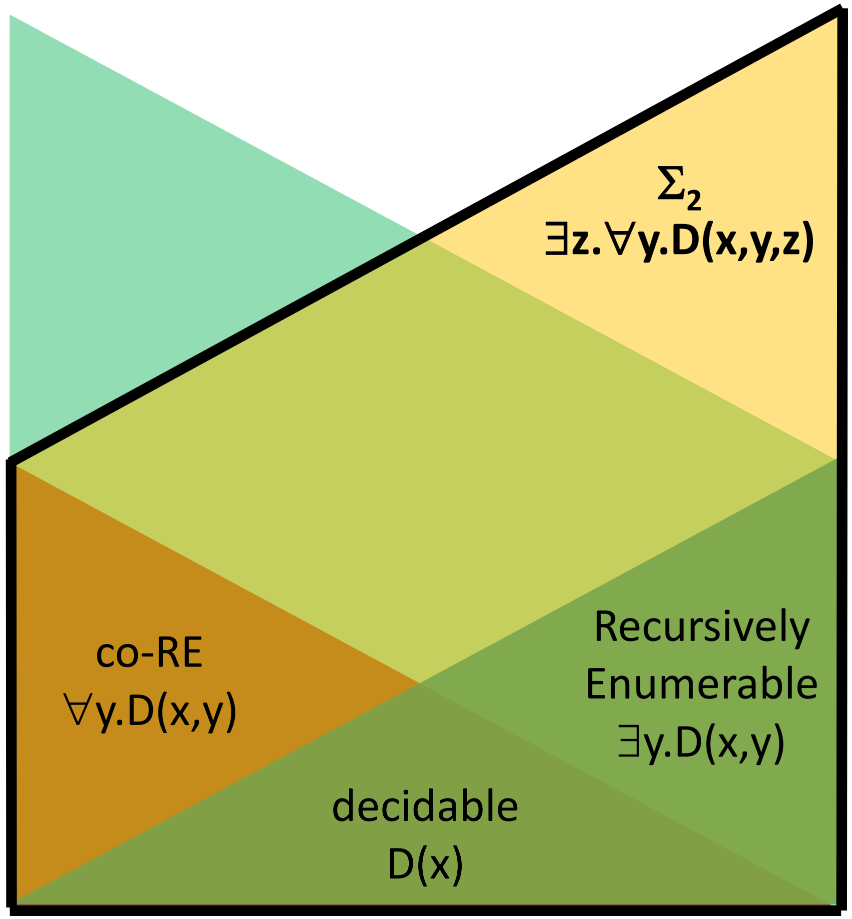
Arithmetic Hierarchy

Example: TM x never halts



Example: TM x eventually halts

Arithmetic Hierarchy



Example: TM x sometimes loops

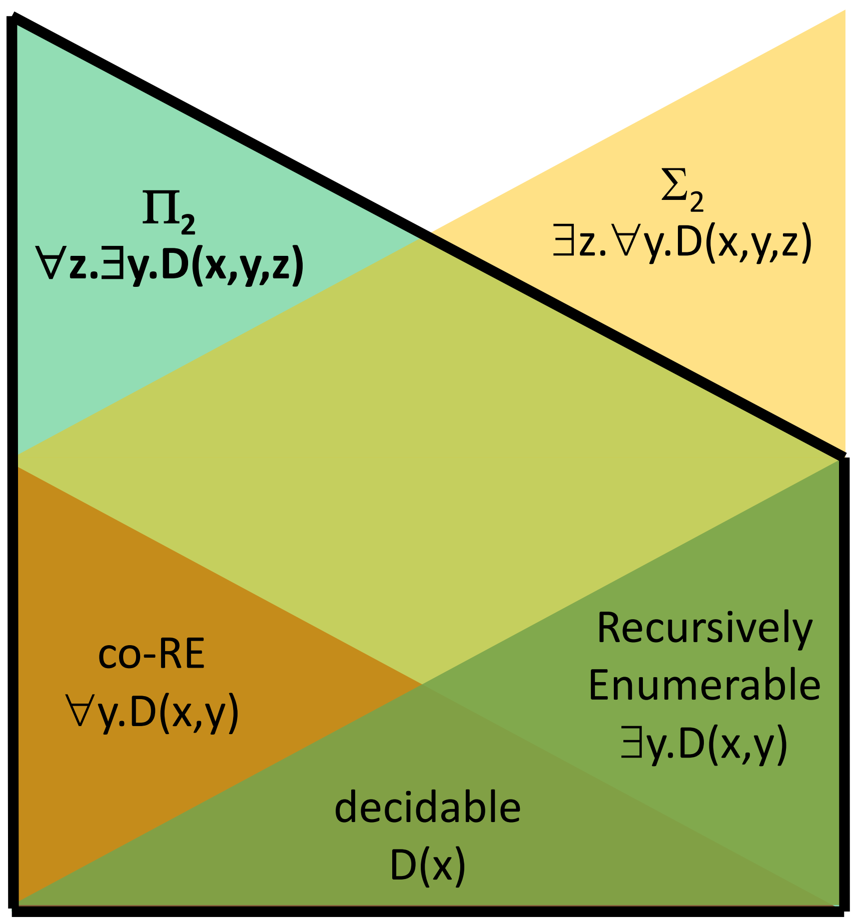
Example: TM x never halts

Example: TM x eventually halts

Arithmetic Hierarchy

Example: TM x always halts

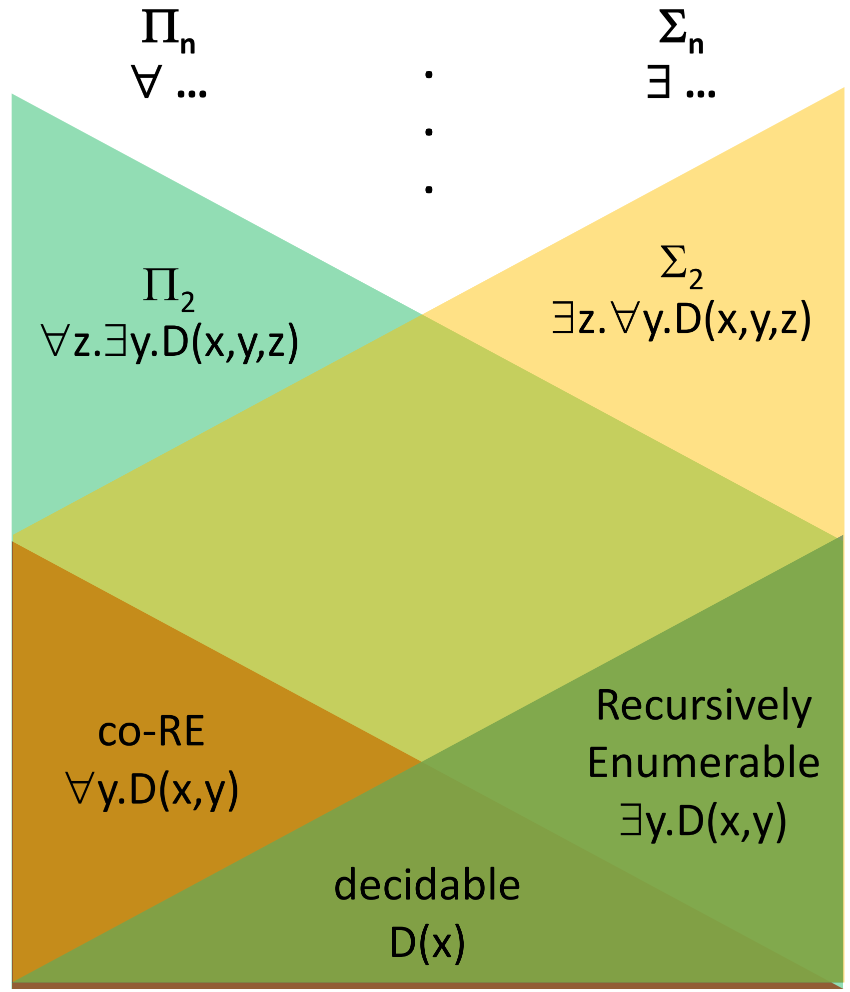
Example: TM x sometimes loops



Example: TM x never halts

Example: TM x eventually halts

Arithmetic Hierarchy



Example: TM x always halts

Example: TM x sometimes loops

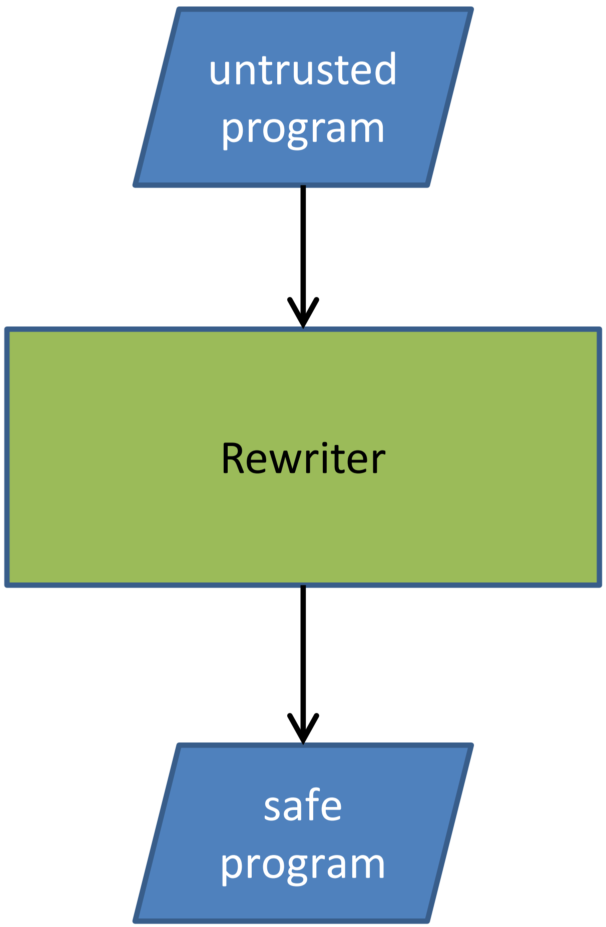
Example: TM x never halts

Example: TM x eventually halts

Computability & Enforceability

- static analysis = recursively decidable
- EM-enforceable = co-RE
- Conclusions so far:
  - EMs are strictly more powerful than static
  - but they cannot enforce RE, higher classes etc.
- What about IRMs? Same as EMs?
  - Surprising answer: No!

IRM Strategy: Rewrite-enforcement



- Approach:
  - transform untrusted code
  - must return new program in finite time
  - transformed code must satisfy policy
  - behavior of safe code must be preserved
- Pros:
  - lowest runtime overhead
  - load-time overhead is once-only
  - sometimes no answer until execution

Rewrite-enforceability

- A policy P is *rewrite-enforceable* if and only if there exists a computable function $R : M \rightarrow M$ such that…
  - $image(R) \subseteq P$ (all outputs are policy-adherent)
  - $P(M) \Rightarrow (R(M) \approx M)$ (behavior of policy-adherent programs is preserved)
- Need a definition of program-equivalence $\approx$
  - turns out any "reasonable" definition will do
  - Example: equal inputs produce equal outputs
- Major difference from EM model: IRM must obey policy, whereas EM has no such obligation
  - IRM's intervention must not be a policy violation
  - IRM must possess an intervention that precludes the impending violation
- On the other hand, IRM has luxury of CHANGING the untrusted code! This is a power that EMs lack.

Main Discoveries

- There are EM-enforceable policies that are not RW-enforceable.
  - Example: Untrusted code must not print the secret stored at address *a*, and must not read address *a*.
- There are RW-enforceable policies that are not EM-enforceable.
  - Example: Untrusted code must behave identically to program M1 on all inputs
- The class of all RW-enforceable policies is not equal to ANY class of the arithmetic hierarchy
  - Open question: What is it, exactly?
  - Some progress: Run-time Enforcement of Nonsafety Policies [Ligatti, Bauer, Walker, TISSEC 2009]
  - See also research on Edit Automata
- Next time:
  - More practical examples of RW-enforceable, non-EM-enforceable policies, and how to enforce them
  - How the theory affects certifying IRM technologies