Software Foundations of Security & Privacy 15315 Spring 2017 Lecture 5: Execution Monitoring, Security Automata

Matt Fredrikson
mfredrik@cs.cmu.edu

January 31, 2017

Runtime enforcement (review)

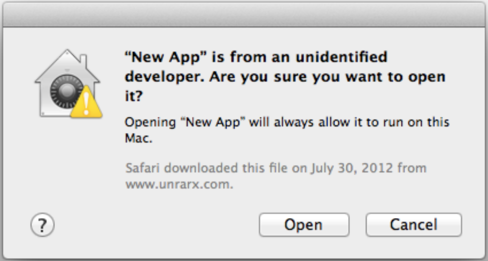
Our focus: policies enforceable by **execution monitors**

Execution Monitor (EM)

An execution monitor is a coroutine that executes in parallel with a **target** program or system.

- Monitor steps of a *single* execution
- Compare observed behavior against a policy
- Terminate the program when policy is violated

Execution monitor examples (review)



- Filesystem access control
- Firewall
- Stack inspection
- Dynamic bounds checking

What's *not* EM? (review)

anything that uses more information than what's available from a single execution of the program/system

In particular, this excludes:

- Information about *future* steps
- Alternative *hypothetical* executions
- *All possible* executions

As we'll see this excludes some important properties

Formalizing execution (review)

A target S is characterized by:

- A set of **atomic actions** A
- A set of sequences Σ_S of elements from A

Sequences in Σ_S can be finite or infinite

What might A look like?

- Set of program states: mappings from *variables* to *values*
- Set of all system calls: `open`, `send`, ...
- Set of primitive commands in server scripting language

Example

How might we model the following program?

Ultimately, our goal is to enforce the policy:

"No `send` after `read`"

```
while(read(&buf, &len, fp)) {
  if(buf[0] == 255)
    send(sock, buf, len);
  printf("%s", buf);
}
```

$$A = \{\mathtt{read}, \mathtt{send}\}$$

$$\Sigma_S = \left\{ \begin{array}{l} [\mathtt{read}, \mathtt{read}, \ldots] \\ [\mathtt{read}, \mathtt{send} \ldots] \\ [\mathtt{read}, \mathtt{send}, \mathtt{read} \ldots] \\ \ldots \end{array} \right\}$$

Formalizing policies (review)

Let Ψ denote the universe of all possible executions in A

- Note: Ψ is not the same as Σ_S
- It contains executions that may not be possible in S
- In particular, $\Sigma_S \subseteq \Psi$

Policy

A **policy** P is a predicate on *sets* of executions. In other words,

$$P \subseteq 2^{\Psi}$$

A target S satisfies P if and only if $\Sigma_S \in P$.

Aside: predicates

A **predicate** f over domain D is a Boolean-valued function:

$$f : D \mapsto \{0, 1\}$$

Predicates specify subsets of their domain

- Let D_f denote the subset of D specified by f
- In set builder notation, D_f is the set:

$$D_f = \{d \in D \mid f(d) = 1\}$$

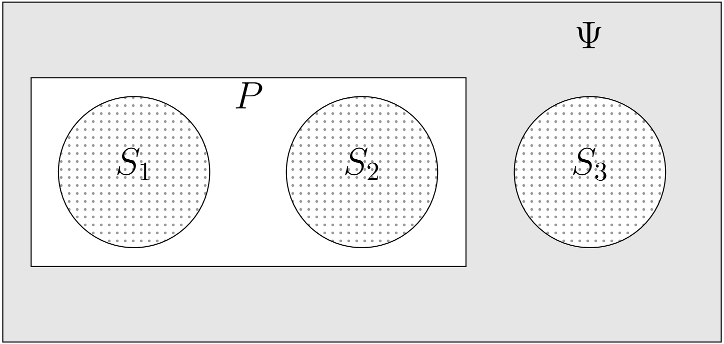
- Often, we don't distinguish between the function symbol f and the set that it represents

So, you can think of a policy P as either

- A subset of the *set of all sets* of executions: $P \subseteq 2^\Psi$
- Or, a Boolean function $P : 2^\Psi \mapsto \{0, 1\}$

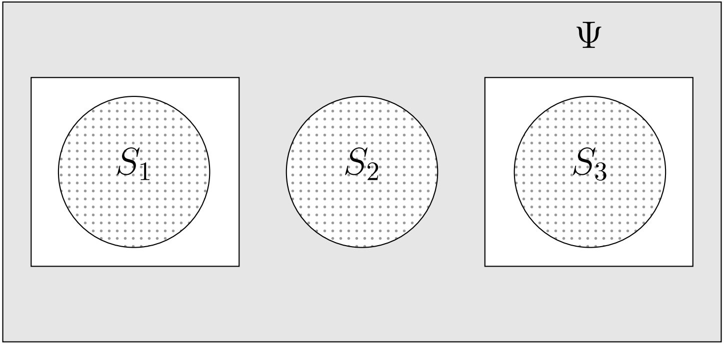
Formalizing policies: intuition

Intuition: a policy is a set whose elements correspond to *systems that are allowed to execute*



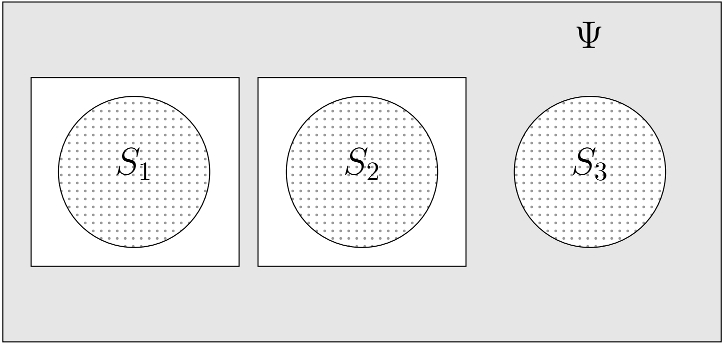
Formalizing policies: intuition

Policies can specify *any* subset of systems



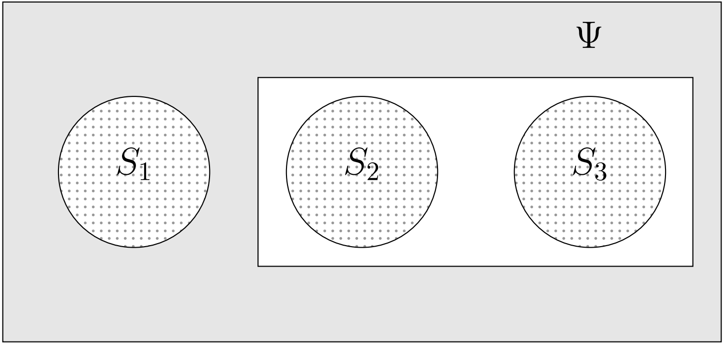
Formalizing policies: intuition

Policies can specify *any* subset of systems



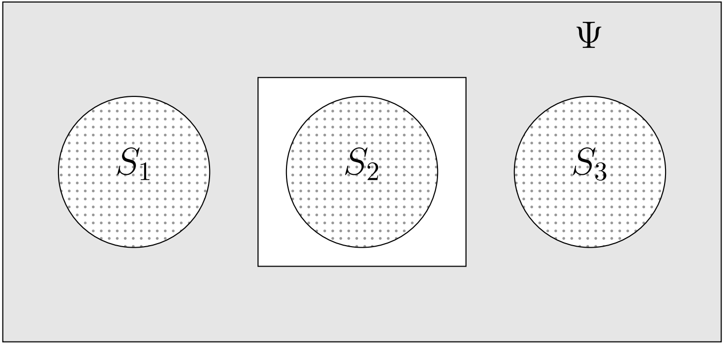
Formalizing policies: intuition

Policies can specify *any* subset of systems



Formalizing policies: intuition

Policies can specify *any* subset of systems



Example

Suppose that we want a simple policy

```
while(read(&buf, &len, fp)) {
  if(buf[0] == 255)
    send(sock, buf, len);
  printf("%s", buf);
}
```

"No `send` after `read`"

$$P = \left\{ \begin{array}{l} \{[\mathtt{read}], [\mathtt{read}, \mathtt{read}], \ldots\} \\ \{[\mathtt{send}], [\mathtt{send}, \mathtt{read}], [\mathtt{send}, \mathtt{send}, \mathtt{read}], \ldots\} \\ \ldots \end{array} \right\}$$

All sets of sequences where `send` only comes after `read`

Formalizing execution monitoring (review)

Recall the key feature of EM:

- Must work by *monitoring the execution of a single execution*

We should be able to express P using simpler means

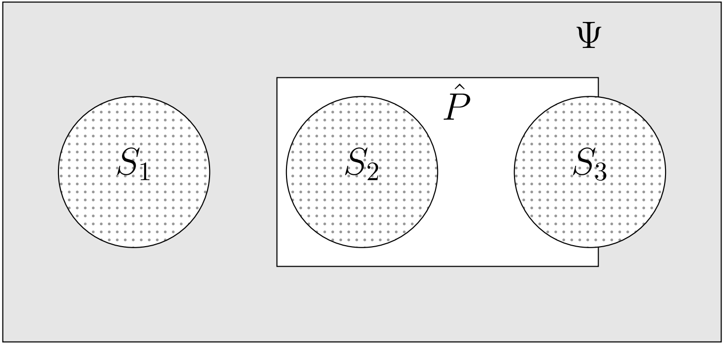
- Let $\hat{P} \subseteq \Psi$ be a set of executions
- We can define $\hat{P}$ so that:

$$\Sigma_S \in P \iff \sigma \in \hat{P} \text{ for all } \sigma \in \Sigma_S$$

Intuition: think of $\hat{P}$ as something like a *regular expression* over executions

Execution monitoring: intuition

Now policies can't necessarily specify any subset of systems



A system is allowed if it is a *subset* of $\hat{P}$

Example

Let's revisit the policy from before

```
while(read(&buf, &len, fp)) {
  if(buf[0] == 255)
    send(sock, buf, len);
  printf("%s", buf);
}
```

"No `send` after `read`"

$$\hat{P} = \left\{ \begin{array}{l} [\mathtt{read}, \mathtt{read}, \ldots], \\ [\mathtt{send}, \mathtt{read}, \ldots], \\ [\mathtt{send}, \mathtt{send}, \mathtt{read}], \\ \ldots \end{array} \right\}$$

We only have a *single* set

Don't need to specify a different set for each allowed system

Which policies are enforceable? (review)

We know they have to be expressible in terms of $\hat{P}$

- i.e., policies determined by each element in Σ_S alone
- These are called **properties**

Are all properties enforceable?

- Recall: can't use information about *future* steps
- So if σ' is a **prefix** of σ and $\sigma' \notin \hat{P}$, but $\sigma \in \hat{P}$
- ...then P isn't enforceable

Prefix closure

Enforceable policies are **prefix-closed**:

- If a trace is in $\hat{P}$ then so are all its prefixes
- If a trace isn't in $\hat{P}$, then none of its extensions are

Prefix closure

Suppose that $\sigma = bad$ is **disallowed**: $\sigma \notin \hat{P}$

- All possible extensions of σ' are also disallowed!
- $badd$, $bada$, $badb$, $badda, \ldots \notin \hat{P}$
- The sequence bad is a **bad thing**
- Once it occurs, there's no way to fix it
- Ex.: leaking password over network, overwriting a file, ...

Suppose that $\sigma = good$ is **allowed**: $\sigma \in \hat{P}$

- All prefixes of σ are also allowed
- g, go, goo, $good$
- But an extension of $good$ **might** be disallowed
- Ex.: `send`,`read` is allowed in policy from before
- But `send`,`read`,`send` is not

More prefix closure

What about the policy:

> "No `send` after `read`, unless `send` is followed by `close`"

This does not have prefix closure

- `read,send,close` is **allowed**
- `read,send` is **disallowed**

To enforce this policy at runtime, we have two options:

- Look into the future to see if the next symbol is `close`
- Wait to see the next symbol after `send` to decide

Why prefix closure is desirable

The “wait and see” approach doesn’t seem too unreasonable

Why do we forbit it by insisting on prefix closure?

After waiting, *the damage might already be done*

- Think of “send after read”: once the data is sent, there’s no taking it back
- We’d like to terminate execute **before** the bad thing happens

With that said, some reasonable policy ideas might need “speculative” enforcement

- Ex.: transactional semantics in assignment 1
- For now, we’re focusing on a simpler notion of enforcement
- Later, we’ll consider more complicated ones

Practical matters

Our goal: define policies that can be enforced *for real*

- We need to know if a policy violation is underway
- We have finite time to wait around for this to happen

Finite refutability

A property P is **finitely refutable** if whenever a trace σ is *not* in $\hat{P}$, there exists some *finite prefix* σ' of σ that is also not in $\hat{P}$.

$$\sigma \notin \hat{P} \Longleftrightarrow \exists i . \sigma[..i] \notin \hat{P}$$

where $\sigma[..i]$ corresponds to the subsequence of σ from its beginning to position i.

Finite refutability

Intuition: We can always write down a **witness** that explains why an execution violates the policy

- The witness is the finite prefix σ'
- The fact that it is finite allows us to write it down

In the "no send after read" example:

- Given `send`,`read`,`send`,`read`,`read`
- `send`,`read`,`send` is the witness
- It ends with the "bad thing", is sufficient to prove violation

More pragmatics

Prefix closure and **finite refutability** simplify matters further:

- We don't need to specify *all allowed executions*
- Instead, specify a set of finite prefixes

These prefixes are "bad things" disallowed by the policy

- Because our policies are properties, we look for bad things in "real time" on a single execution
- By prefix closure, once we see the bad thing happen, we know the policy is permanently violated
- By finite refutability, if a policy violation happens we will detect it in finite time
- Plus, we can give a witness to prove that it was violated

Enforceable policies

Properties satisfying prefix closure and finite refutability are called **safety properties**

What policies are safety properties?

- **Access control**, defined broadly as policies that proscribe unacceptable operations. This includes filesystem permissions, bounds checking, read-xor-execute, ...
- **Information flow** is *not* safety: it cannot be defined in terms of individual executions. Did we define information flow with "no send after read"?
- **Availability** is *not* safety: any partial execution can be *extended* to grant access to the resource in question, so we can't define a set of finite prefixes to characterize availability.

Safety and information flow

Before, we actually enforced
"no send after read"

Ideally, we wanted to prevent:

$$\texttt{fp} \longrightarrow \texttt{sock}$$

How is our *actual* policy *not* the same?

This policy *approximates* information flow

- Prevents flow from happening
- Also prevents other things

```
while(read(&buf, &len, fp)) {
  if(buf[0] == 255)
    send(sock, buf, len);
  printf("%s", buf);
}
```

```
while(read(&buf, &len, fp)) {
  memset(buf, 0, len);
  send(sock, buf, len);
  printf("%s", buf);
}
```

Does this flow `fp` to `sock`?

Information flow isn't EM-enforceable

Suppose x and y are bits

```
if(x)
  y = 0;
else
  y = 1;
```

With executions:

$$\left\{\begin{array}{c} [(x \mapsto 0, y \mapsto 0), (x \mapsto 0, y \mapsto 1)] \\ [(x \mapsto 0, y \mapsto 1), (x \mapsto 0, y \mapsto 1)] \\ [(x \mapsto 1, y \mapsto 0), (x \mapsto 1, y \mapsto 0)] \\ [(x \mapsto 1, y \mapsto 1), (x \mapsto 1, y \mapsto 0)] \end{array}\right\}$$

What *is* information flow from x to y?

Changes to x cause changes in y

A thought experiment

Let S_1:

```
if(x)
  y = 0;
else
  y = 1;
```

And S_2:

```
x, y = 0, 1;
```

And S_3:

```
x, y = 1, 0;
```

What's going on?

- S_1 flows information from x to y
- S_2, S_3 do not
- Together, S_2, S_3 can replicate all executions of S_1

$$\left\{ \begin{array}{c} [(x \mapsto 0, y \mapsto 0), (x \mapsto 0, y \mapsto 1)] \\ [(x \mapsto 0, y \mapsto 1), (x \mapsto 0, y \mapsto 1)] \\ [(x \mapsto 1, y \mapsto 0), (x \mapsto 1, y \mapsto 0)] \\ [(x \mapsto 1, y \mapsto 1), (x \mapsto 1, y \mapsto 0)] \end{array} \right\}$$

A thought experiment

Let S_1:

```
if(x)
  y = 0;
else
  y = 1;
```

And S_2:

```
x, y = 0, 1;
```

And S_3:

```
x, y = 1, 0;
```

Experiment as follows:

1. You pick an initial value for x, show it to me.
2. I (secretly) pick $i \in \{1, 2, 3\}$, run S_i on your input to get execution σ.
3. You see σ, try to guess i.

Suppose you pick $x = 0$ initially

Now I show you:

$$\sigma = [(x \mapsto 0, y \mapsto 0), (x \mapsto 0, y \mapsto 1)]$$

How to distinguish between S_1 and S_2?

A (modified) thought experiment

Let S_1:

```
if(x)
  y = 0;
else
  y = 1;
```

And S_2:

```
x, y = 0, 1;
```

And S_3:

```
x, y = 1, 0;
```

New experiment as follows:

1. You pick **two** initial values for x.
2. I (secretly) pick $i \in \{1, 2, 3\}$, run S_i on your inputs.
3. You see the executions σ_1, σ_2, try to guess i.

Now you win every time

Hyperproperties

Information flow is a *hyperproperty*

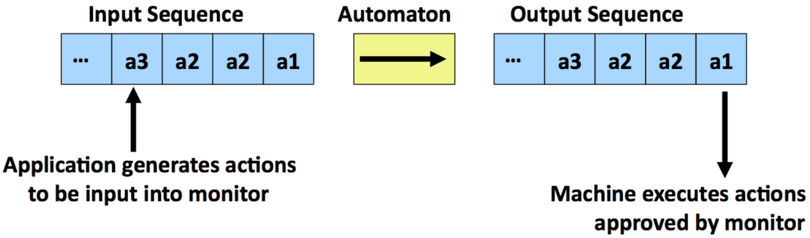
In particular, it is *2-safety*:

- Finitely refutable over *pairs* of traces

Can generalize to k-safety

- Lots of interesting properties...
- Quantitative privacy
- Statistical availability

Security automata



- Formal model of an execution monitor
- "Language" for specifying policies
- Corresponds to $\hat{P}$ from before

Image credit: Lujo Bauer

Security automata

Security automaton

A **security automaton** is a non-deterministic finite or infinite-state automaton defined by:

- Q: a countable set of **automaton states**
- $Q_0 \subseteq Q$: a countable set of **initial states**
- I: a countable set of **input symbols**
- $\delta : (Q \times I) \mapsto 2^Q$: a **transition function**

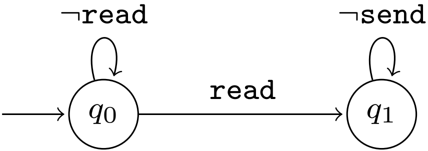
Security automata: semantics

Notice: no accepting states

Let Q' be the *current states*

To process execution $s_1 s_2 \ldots$:

1. Read the next input symbol s_i
2. Change Q' to
$$\bigcup_{q \in Q'} \delta(q, s_i)$$
3. If Q' ever becomes empty, the input is rejected



An action is allowed if a transition exists for it

Can process both finite and infinite sequences!

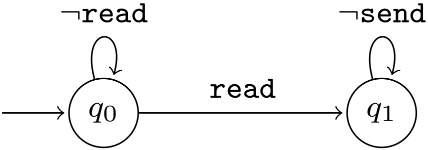
Security automata: input symbols

We label edges with *transition predicates*

- Boolean-valued and total
- Domain: I

Let p_{ij} label edge between states i, j

- p_{ij} specifies a **subset of** I
- $p_{ij}(s)$ is **satisfied** if s is in that subset
- e.g., $\neg$`read` is satisfied by any symbol except `read`



Example

`Goal`: enforce array bounds checking on `a`

- States correspond to $|\texttt{a}|$
- Notice: infinite set of states if $|\texttt{a}|$ is unbounded
- Input symbols I:
  - $\texttt{malloc}(n)$
  - `free`
  - $\texttt{read}(i)$
  - $\texttt{write}(i, v)$

  for all integers n, i, v

Transitions:

- On $\texttt{malloc}(n)$:
  $(|\texttt{a}| = n') \longrightarrow (|\texttt{a}| = n)$
- On `free`:
  $(|\texttt{a}| = n') \longrightarrow (|\texttt{a}| = 0)$
- On $\texttt{read}(i)$:
  $(|\texttt{a}| = n) \longrightarrow (|\texttt{a}| = n)$
  if $0 \leq i < n$
- On $\texttt{write}(i, x)$:
  $(|\texttt{a}| = n) \longrightarrow (|\texttt{a}| = n)$
  if $0 \leq i < n$

Using security automata for enforcement

Security automata as the foundation of an execution monitor:

1. Initialize automaton on program/system startup
2. **Before** the target executes a step, generate the corresponding symbol
3. If the automaton can make a transition, let the target execute the step
4. If the automaton can't transition, terminate the target

This allows termination on *attempted* violation

Assumption: bounded memory

We allowed automata to have (countably) infinite states

This is necessary for recognizing certain safety properties

- Whether a prefix should be rejected might depend on every symbol in the prefix
- The amount of memory needed to remember the past grows without bound

In practice, most security policies don't need this

- Restricting the automaton to a finite set of states is probably fine for most purposes

Assumption: target control

Need ability to terminate the target on policy violation

This makes certain safety properties non-enforceable

Real-time availability

One principal cannot be denied use of a resource for more than M seconds.

Safety characterization: “Bad thing” is an unavailable interval spanning more than M seconds.

Passage of real time is an input symbol

Monitor cannot exert control over passage of time!

Assumption: mechanism integrity

To correctly enforce a policy, we must assume:

- Input symbols correspond to the actual execution
- Transitions correspond to the automaton's true transition function

If target corrupts mechanism, it can violate these assumptions

Address this with two strategies

- **Isolation**: target must be unable to write to the internal representation of the automaton
- **Complete mediation**: make sure that all aspects of execution that might generate input symbols are covered by implementation

Proving correct enforcement

Goal: Show that when S executes under enforcement of SA P,

- S terminates when its execution violates P
- S continues to execute otherwise

This requires a proof that the implementation satisfies:

1. Complete mediation
2. Target control
3. Isolation

We'll see how different implementation strategies lead to different kinds of proof

More pragmatics

Two mechanisms are needed to implement SA:

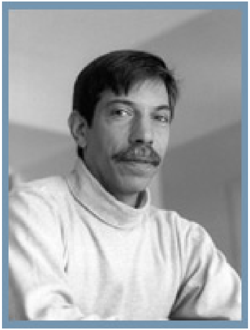
- **Input Read:** Determines that an input symbol has been produced by the target, forwards that symbol to the automaton simulation
- **Transition:** Determines whether the automaton can make a transition on a given input symbol, and if so, executes that transition by updating automaton state appropriately.

These implementations affect correctness and performance

Further reading

Enforceable Security Policies

FRED B. SCHNEIDER
Cornell University



Further reading

Recognizing safety and liveness *

Bowen Alpern[1] **and Fred B. Schneider**[2]

[1] IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, USA
[2] Department of Computer Science, Cornell University, Ithaca, NY 14853, USA