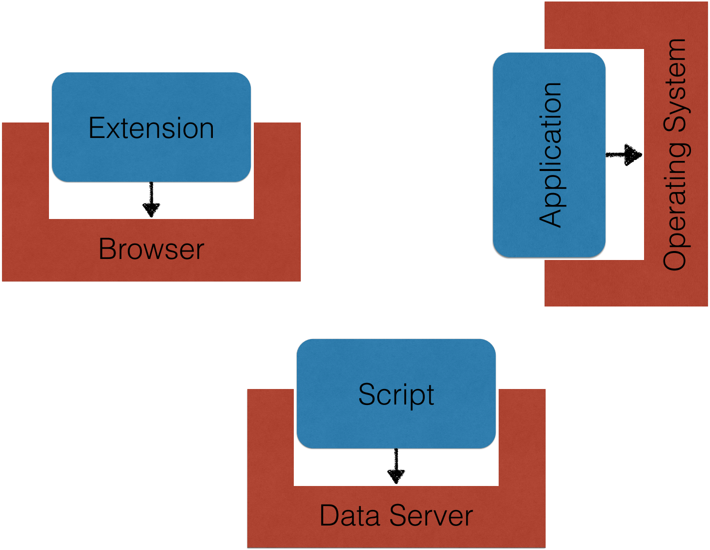
Software Foundations of Security & Privacy 15315 Spring 2017 Lecture 4: Enforceable Security Policies

Matt Fredrikson
mfredrik@cs.cmu.edu

January 26, 2017

Securing Extensible Systems



Key Questions

What security policies can we enforce?

- Topic of today's lecture

What mechanisms can we use?

Key Questions

What security policies can we enforce?

- Topic of today's lecture

What mechanisms can we use?

- Type checking
- Static verification
- Program rewriting
- **Runtime enforcement**

Runtime enforcement

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

Runtime enforcement

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

Execution monitor examples

- Filesystem access control
- Firewall
- Stack inspection
- Dynamic bounds checking
- Malware detectors
- Chrome's *Content Security Policies* (CSP)
- ...

What's *not* EM?

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

What's *not* EM?

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

What's *not* EM?

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

What's *not* EM?

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

Later, we'll talk about techniques that aren't limited in this way

- Verifying compilers, type systems
- Anything classified as "static analysis"

Formalizing execution

A target S is characterized by:

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

Formalizing execution

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

Formalizing execution

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?

Formalizing execution

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*

Formalizing execution

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`, ...

Formalizing execution

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 do we model the following program?

$$
\textbf{while}(x < y)\ \{ \\
\quad x := x + 1; \\
\}
$$

Example

How do we model the following program?

while$(x < y)$ $\{$
  $x := x + 1;$
$\}$

Atomic actions $A = \{$mappings from x, y to $\mathbb{Z}\}$

Example

How do we model the following program?

while$(x < y)$ $\{$
  $x := x + 1;$
$\}$

Atomic actions $A = \{$mappings from x, y to $\mathbb{Z}\}$

$$\Sigma_S = \left\{ \begin{array}{l} \dots \\ [(x \mapsto -1, y \mapsto 0), (x \mapsto 0, y \mapsto 0)] \\ [(x \mapsto 0, y \mapsto 0)] \\ [(x \mapsto 0, y \mapsto 1), (x \mapsto 1, y \mapsto 1)] \\ [(x \mapsto 0, y \mapsto 2), (x \mapsto 1, y \mapsto 2), (x \mapsto 2, y \mapsto 2)] \\ \dots \end{array} \right\}$$

Example

How might we model the following program?

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

Example

How might we model the following program?

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

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

Example

How might we model the following program?

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

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

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

Formalizing policies

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, $\Sigma_S \subseteq \Psi$

Formalizing policies

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 $\Sigma_S \in P$.

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`"

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`"

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

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`"

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

Does the program satisfy this policy?

Formalizing execution monitoring

Recall the key feature of EM:

- Must work by monitoring the execution of a single execution

Formalizing execution monitoring

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:

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

Formalizing execution monitoring

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:

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

Why is this simpler?

Formalizing execution monitoring

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:

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

Why is this simpler?

- We don't need to specify all allowed systems to get P!
- We can just specify all allowed *executions* instead.

Formalizing execution monitoring

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:

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

Why is this simpler?

- We don't need to specify all allowed systems to get P!
- We can just specify all allowed *executions* instead.
- Again, why is this simpler?

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`"

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`"

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

Which policies are enforceable?

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

Which policies are enforceable?

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?

Which policies are enforceable?

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

Which policies are enforceable?

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 $\sigma \in \hat{P}$

Which policies are enforceable?

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

Which policies are enforceable?

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

Practical matters

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

Practical matters

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

- Ultimately, we'd like to know if a policy violation is underway
- We have finite time to wait around for this to happen

Practical matters

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

- Ultimately, we'd like 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.

More pragmatics

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

More pragmatics

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

- We don't need to specify *all allowed executions*

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

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

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

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 (in principle) detect it

Enforceable policies

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

What policies are safety properties?

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, ...

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.

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"?

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 enforced
"no send after read"

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

Safety and information flow

Before, we enforced "no send after read"

We wanted to prevent:

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

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

Safety and information flow

Before, we enforced "no send after read"

We wanted to prevent:

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

How is this *not* an information flow policy?

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

Safety and information flow

Before, we enforced "no send after read"

We wanted to prevent:

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

How is this *not* an information flow policy?

```
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, , len);
  printf("%s", buf);
}
```

Does this flow `fp` to `sock`?

Safety and information flow

Before, we enforced "no send after read"

We wanted to prevent:

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

How is this *not* an information flow policy?

This policy *approximates* information flow

- Prevents a 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, , 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:

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

Information flow isn't EM-enforceable

Suppose x and y are bits

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

What *is* information flow from x to y?

With executions:

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

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;
```

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.

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.
2. I (secretly) pick $i \in \{1, 2, 3\}$, run S_i on your input.

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.
2. I (secretly) pick $i \in \{1, 2, 3\}$, run S_i on your input.
3. You see the execution σ, try to guess i.

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.
2. I (secretly) pick $i \in \{1, 2, 3\}$, run S_i on your input.
3. You see the execution σ, try to guess i.

Suppose I give you:

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

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;
```

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.

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;
```

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*

Hyperproperties

Information flow is a *hyperproperty*

In particular, it is *2-safety*:

- Finitely refutable over *pairs* of traces

Hyperproperties

Information flow is a *hyperproperty*

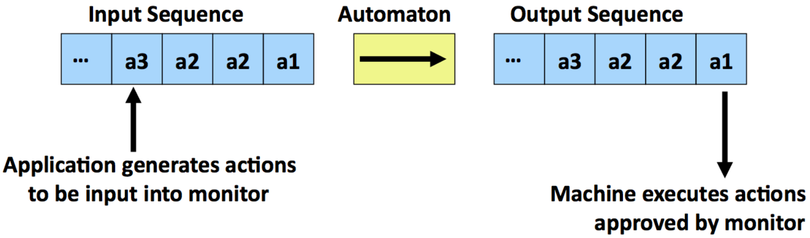
In particular, it is *2-safety*:

- 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

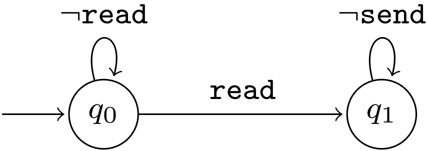
Security automaton

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**
- A: a countable set of **input symbols**
- $\delta : (Q \times I) \mapsto 2^Q$: a **transition function**

Security automata: semantics

Notice: no accepting states

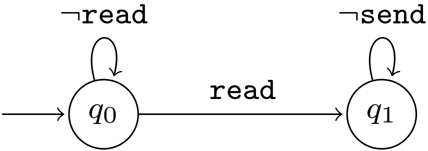


Security automata: semantics

Notice: no accepting states

Let Q' be the *current states*

To process execution $s_1 s_2 \ldots$:



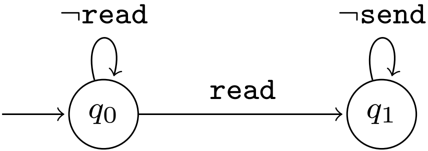
Security automata: semantics

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



Security automata: semantics

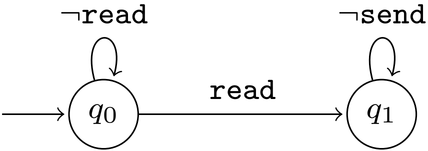
Notice: no accepting states

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

$$\bigcup_{q \in Q'} \delta(q, s_i)$$



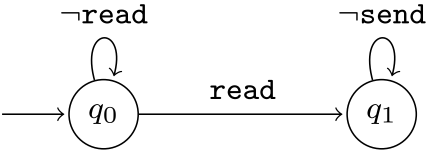
Security automata: semantics

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



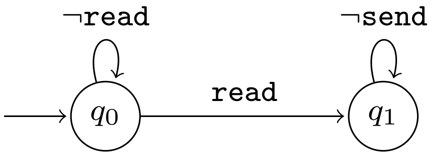
Security automata: semantics

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

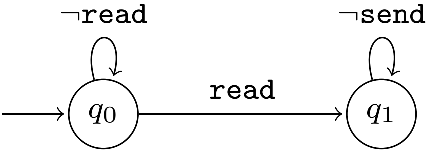
Security automata: semantics

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



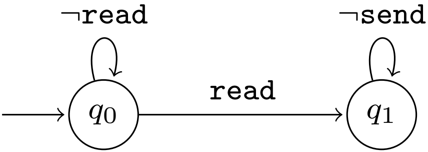
An action is allowed if a transition exists for it

Can process both finite and infinite sequences!

Security automata: input symbols

We label edges with *transition predicates*

- Boolean-valued and total
- Domain: A



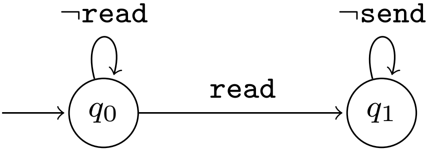
Security automata: input symbols

We label edges with *transition predicates*

- Boolean-valued and total
- Domain: A

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

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



Using security automata for enforcement

Security automata can be implemented to form the basis 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

Assumption: bounded memory

We allowed automata to have (countably) infinite states

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

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

Assumption: target control

Need ability to terminate the target on policy violation

This makes certain safety properties non-enforceable

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.

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 time cannot be stopped!

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

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

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

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

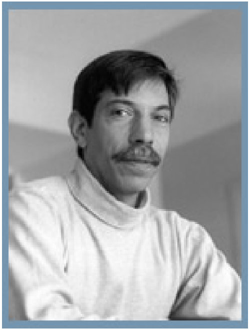
Address this with two strategies

- **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

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