
SECURITY
POLICIES
AND THEIR
ENFORCEMENTS

Secure Programming Languages

Stack, Heap and code area

- Stack Canaries
- Reorder layout of AR and Heap elements (randomization)
- Shadow stack
- NX memory
- Enclaves
- FAT Pointer & Data Descriptor
- Web Assemby

Static Analysis

- Later

Next step: security policies

Introduce programming abstractions for

1) Defining security policies

2) Implementing enforcement mechanisms

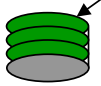
Program Execution

Monitor

Multimedia

Program

Network



Disk

Memory

Intelligent Systems

Program is not considered in isolation. We have devices and peripherals working with it.

Security Policies

In the context of **language-based security**, **security policies** define the rules and constraints that govern how programs can access and manipulate data to prevent security vulnerabilities.

Example of security policies are confidentiality, integrity, and availability

↳ Avoid data unavailability to non authorized individuals

Another policy is integrity

Our aim is to enforce security policies by leveraging programming language features.

Enforcement Mechanisms

Enforcing security policies at the language level involves **dynamic** and **static** techniques

Dynamic Enforcement

Runtime Monitoring: Checking policy adherence during execution (e.g., sandboxing)

Enforcement Mechanisms in VM: Virtual machines (e.g., WebAssembly, JVM) enforce security through restricted execution environments. *

EXECUTION

Reference Monitors: Intercepts security-sensitive operations and ensures they comply with policies. ACTIVE ENTITIES

* In JVM you have runtime metadata for array, for instance. Protection against buffer OF.

Static enforcements

Type Systems & Type Safety: Ensuring variables and functions adhere to predefined types to prevent unintended access or corruption. THIS IS DONE STATICALLY

Static Analysis: Using tools like data flow and control flow analysis to detect security violations before execution.

Formal Verification: Applying mathematical proofs to ensure compliance with certain security policies.

DYNAMIC ENFORCEMENT

Some challenging questions

Can we prove that mechanism **M** enforces the security policy **P**?	Are there limits to what is enforceable?	Policies?	OVERALL
• What is the mathematical definition of a policy? • What is the programming abstractions for declaring security policies? • What does it mean to enforce a policy within a programming language?	• Which enforcement approaches are best suited to which	• Are there some policies that are completely beyond any known enforcement strategy? ① • Are some enforcement approaches strictly more powerful than others?	• what is the landscape of policies, policy classes, and enforcement mechanisms?

① Can we characterize which kind of properties a sec. policy can express?

Starting point

Enforceable Security Policies

F. Schneider, ACM Transactions on Information and System Security, February 2000

Abstract

A precise characterization is given for the class of security policies enforceable with mechanisms that work by monitoring system execution, and automata are introduced for specifying exactly that class of security policies. Techniques to enforce security policies specified by such automata are also discussed.

Example of runtime mechanism that looks at the behavior of another program

Execution Monitor

Execution Monitors (EMs)

- EMs watch un-trusted programs at runtime
- Events (raised by the run time executions) are mediated by the EM
- Violations solicit EM interventions (e.g. termination)

Example: File system access control

- EM is inside the OS
- decides policy violations using access control lists (ACLs)

Execution Monitor

- EMs are run-time modules that runs in parallel with an application
  - Tracks execution flow at runtime to **detect and prevent** security violations dynamically.
  - monitor decisions may be based on **execution history**

Program Execution Revisited

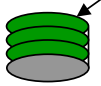
Monitor

Multimedia

Program

Execution Monitor

Network



Disk

Memory

Intelligent Systems

The context is the same as before, but a monitor "encloses" what the program is doing.

The Ideal Execution Monitor

1. **Sees *everything* a program is about to do before it does it**
2. **Can *instantly* and *completely* stop program execution (or prevent action)**
3. **Has *no other effect* on the program or system**

↳ as soon as we realise execution violates policy without overhead

↳ part of the program that do not cause problems as not affected by it overhead wise

The Ideal Execution Monitor

1. **Sees *everything* a program is about to do before it does it**
2. **Can *instantly* and *completely* stop program execution (or prevent action)**
3. **Has *no other effect* on the program or system**

Can we implement this?

Probably not …….

Real ~~Ideal~~ Execution Monitor

1. **Sees ~~*everything*~~ most things a program is about to do before it does it**
2. **Can ~~*instantly* and *completely* stop program execution (or~~ prevent action~~)~~**
3. **Has ~~*no other*~~ limited *effect* on the program or system**

1. Does not necessarily see EVERYTHING. Can see most of them.
2. Cannot terminate instantly as soon as there's a vulnerability exploited.
3. Execution has still some effect on program. If is compositional, yes, but you pay in terms of performance. ! ?

Operating Systems

- Provide execution monitors for most security-critical resources
  - When a program opens a file (in Unix, Linus, Windows), the OS checks that the principal running the program can open that file

through the use of ACL.

What does EXECUTION MONITOR do?

EM Operating systems (1)

1. **Policy Enforcement**
   - Ensures that processes comply with predefined security policies (e.g., access control rules, privilege separation).

↳ Policy is enforced at runtime

Runtime Decision Making

- Continuously monitors process execution and decides whether to allow, deny, or log specific actions.

EM Operating systems (2)

Containment and Isolation

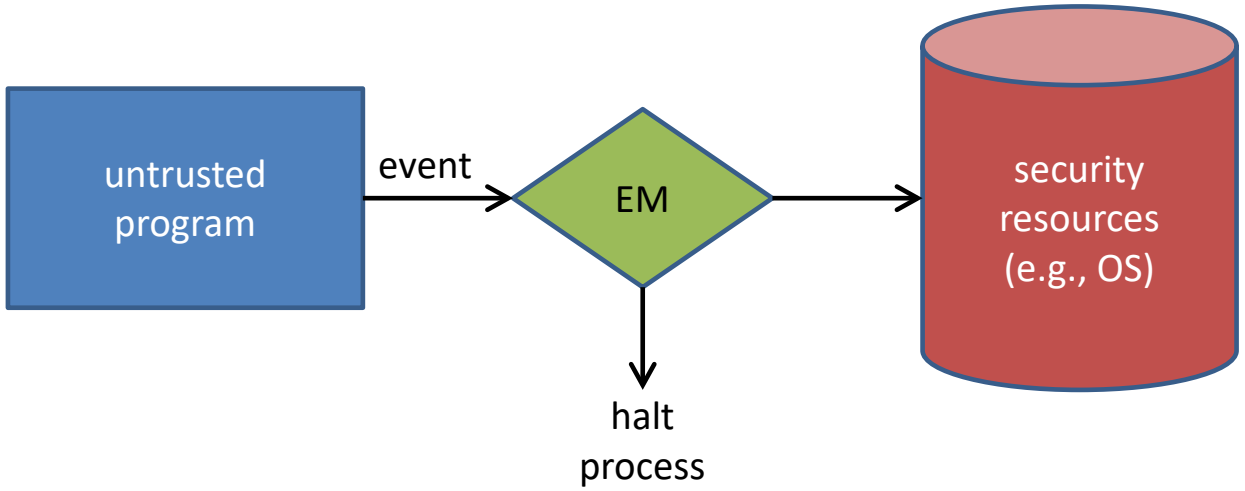
- Restricts the behavior of untrusted applications to prevent unauthorized resource access.

Intrusion Detection & Prevention

- Identifies and blocks malicious activities (e.g., unauthorized file access or privilege escalation).

OS Execution Monitor

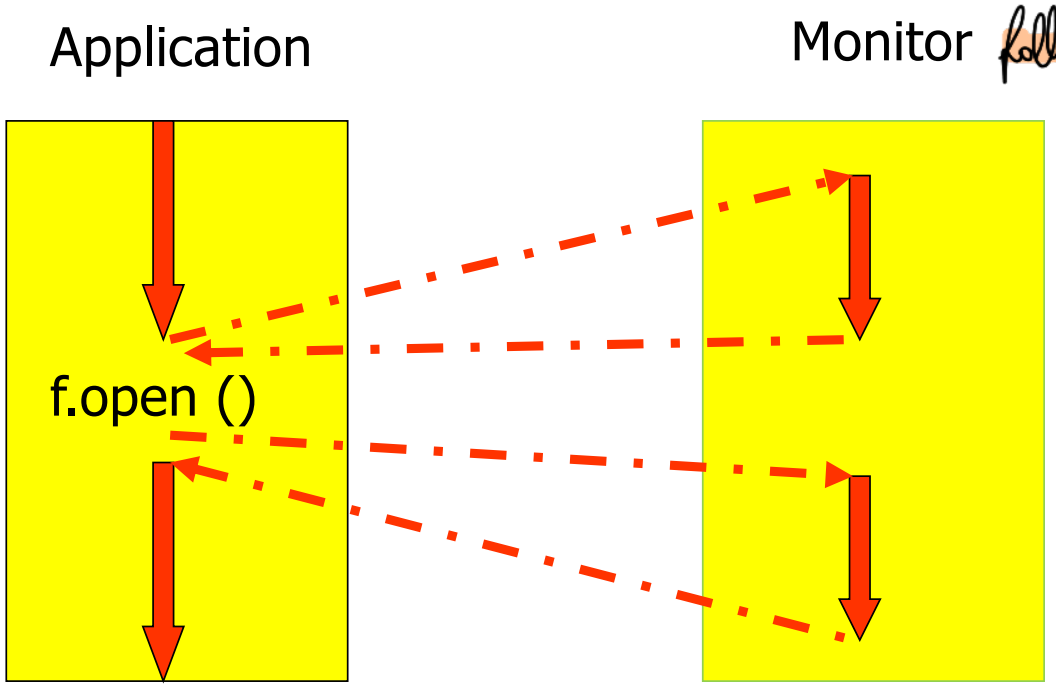
OS execution monitors typically behave as **monitors** that mediate access to system resources.



Execution Monitor

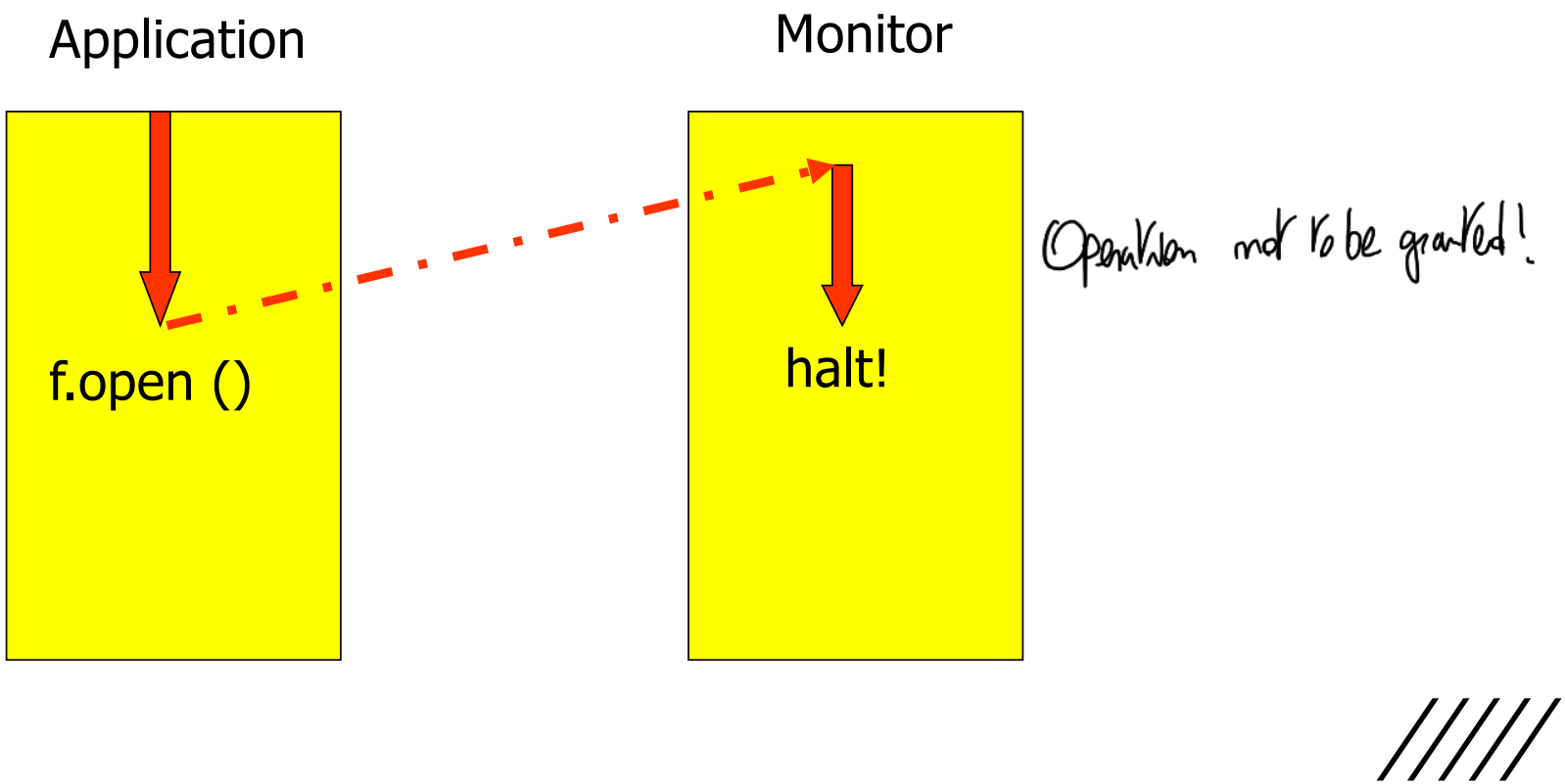
- EMs are run-time modules that runs in parallel with an application
  - Tracks execution flow at runtime to **detect and prevent** security violations dynamically.
  - monitor decisions may be based on **execution history**

EM: Good Operations



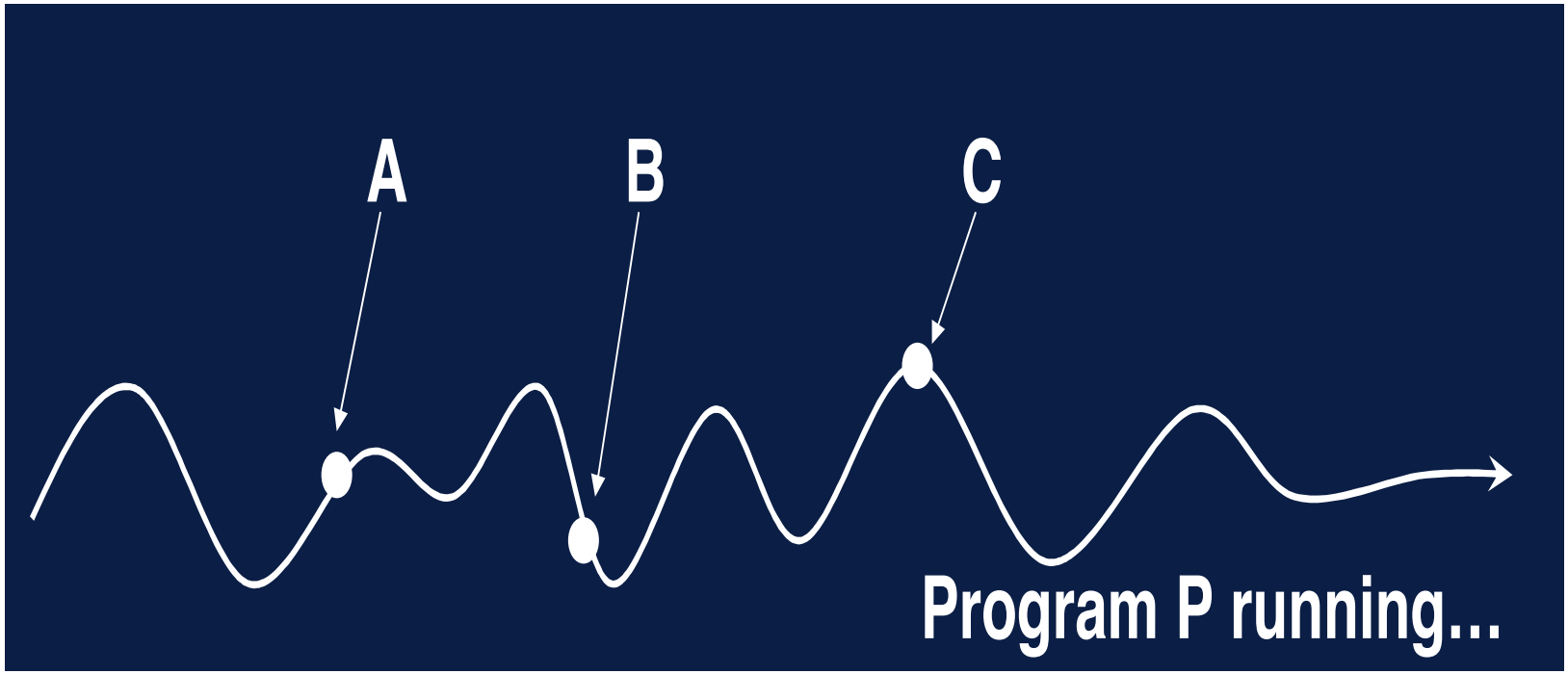
follows a certain control flow.
If the app does not affect
a security relevant resource
the monitor does not intervene

EM: Bad Operation

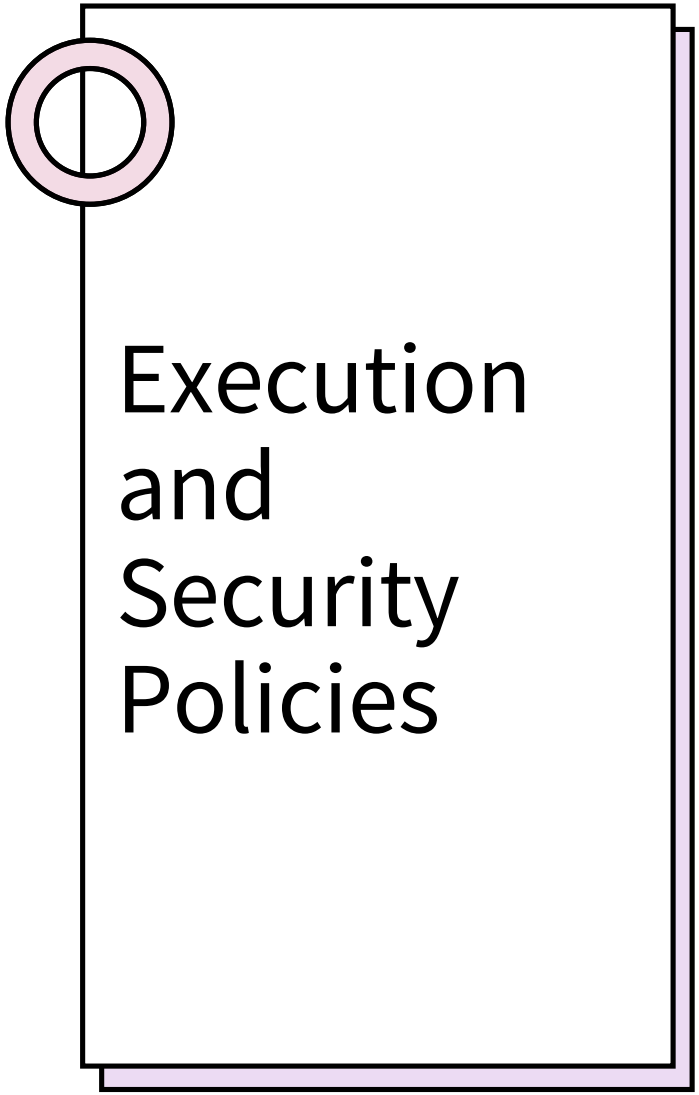


Execution flow: intuition

program's execution on a given input as a sequence of runtime events



Behaviour of a program is seen as a sequence of events. Events might not be relevant to security (not considered by monitor). The others are visible and analysed by EM.



Execution and Security Policies

What's a program (at run-time)?

- A set of possible executions

What's an execution?

- A sequence of states

What's a security policy?

- A predicate on a set of executions

① A program is a set of possible sequences of events.
But what is a sequence? The execution:
② A sequence of states [At the level of formal methods, the sequence of code]
③ is a property over a set of executions.

Definitions and Notations

- An *execution (or trace)* **s** is a sequence of security-relevant program *events* **e** (also called actions)
- Sequence may be **finite** or **(countably) infinite**
  - **$s = e_1; e_2; \ldots; e_k ; e_{halt}$**
  - **$s = e_1; e_2; \ldots; e_k; \ldots$**
  - The empty sequence ε is an execution
  - If s is the execution **$e_1; e_2; \ldots; e_i; \ldots e_l; \ldots$** then **s[i]** is the execution **$e_1; e_2; \ldots; e_i;$**
- We simplify the formalism.
  - We model program termination as an infinite repetition of **e_{halt}** event.
  - Result: now all executions are infinite length sequences

Definitions and Notations (cont.)

All possible behaviors

- A program **S** is a **set** of sequences (possible executions)
  - A program is modelled as the set S = {s_1, s_2, …..}
- A security policy **P** is a **property** of programs: is a math formula representing property we would expect from program execution.
- A policy partitions the program space into two groups:
  - Permissible
  - Impermissible

  } Disjoint sets of sequences
- Impermissible programs are censored somehow (e.g.,terminated on violating runs)

Formally...

- Ψ : set of all possible executions (can be infinite)
- Σ_S: set of executions possible by target program S ↗ sets of execution of program S
- P: security policy

  set of executions → Boolean

  funct. Verifying a set of execution. Returns true if property is satisfied otherwise false

$Program\ S$ is secure wrt the security policy P iff $P\ (\Sigma_S)$ is true.

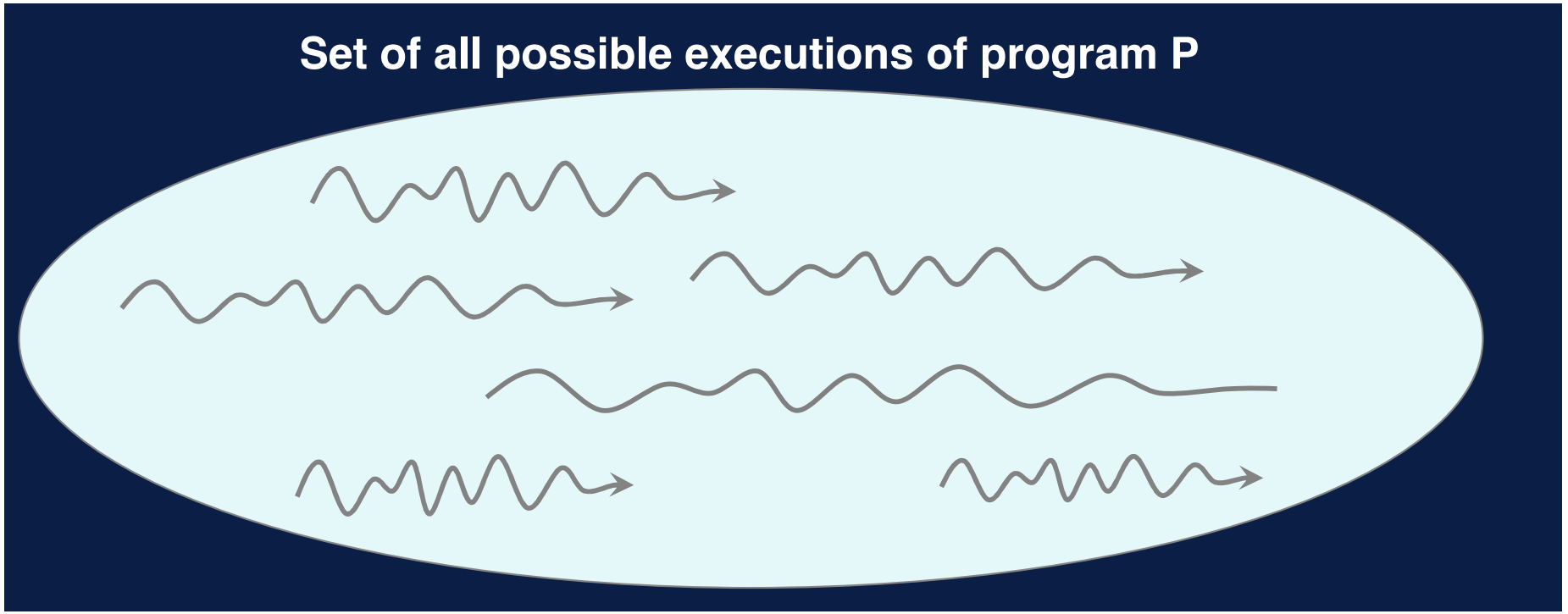
↓ All the sequences inside satisfy property.

So set of non admissible sequences is empty. there is NO execution violating property.

////

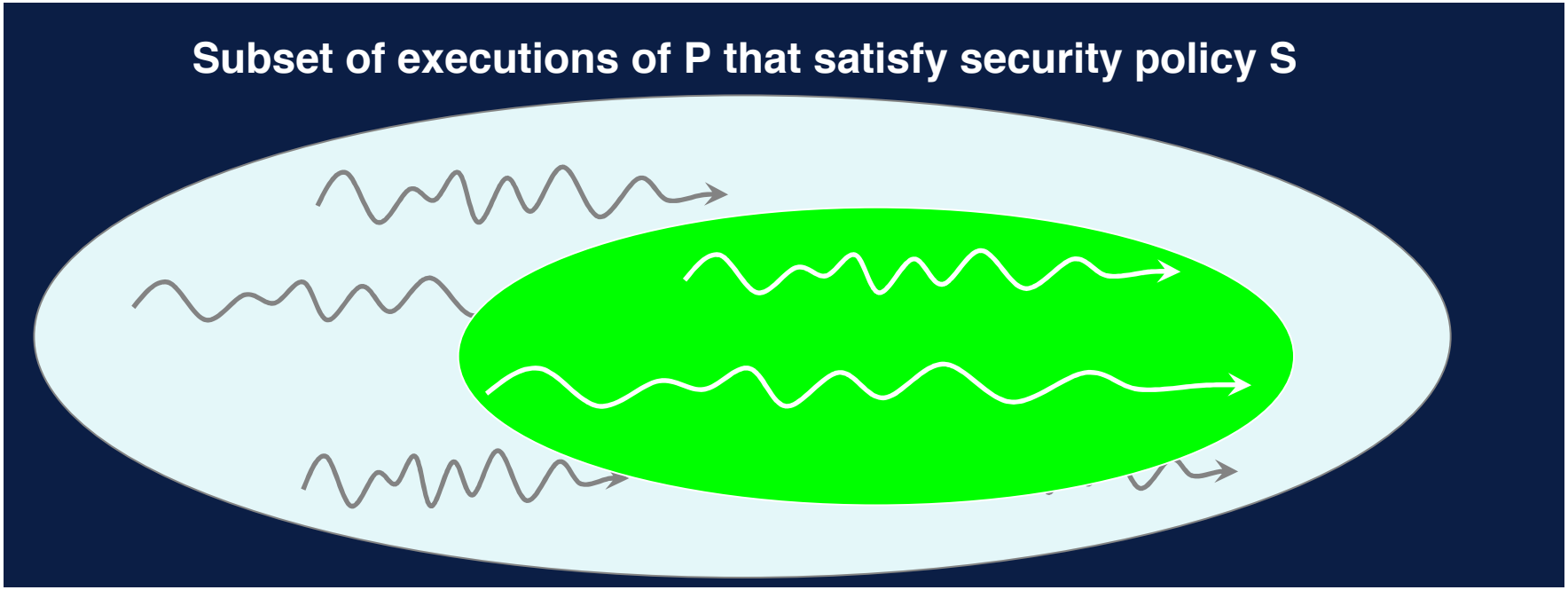
P(generic set) is an operation we can do.

Execution Traces



Security Policies

Program is not secure with respect to the policy.



But are we able to check all possible executions?

Security Policies: Examples

- **Access Control policies** specify that no execution may operate on certain resources such as files or sockets, or invoke certain system operations.
- **Availability policies** specify that if a program acquires a resource during an execution, then it must release that resource at some (arbitrary) later point in the execution. DoS violate this policy.
- **Bounded Availability policies** specify that if a program acquires a resource during an execution, then it must release that resource by some fixed point later in the execution

Execution Monitors cannot enforce all Security Policies

SP difficult to enforce: a policy that should be capable to predict what happens in the future of execution.

- Some policies depend on:
  - Knowing about the future
    - If the program charges the credit card, it must eventually ship the goods
  - Knowing about all possible executions
    - Information flow – can't tell if a program reveals secret information without knowing about other possible executions
- Execution Monitors can only know about past of this particular execution

EM-enforceable policies : define enforcement mechanisms. We define EM enforcement mechanism that enforces a policy.

1. EM policies are universally quantified predicates over executions
   - $\forall s. P(s)$
   - **Policy P is called the detector.**

Policies universally hold

First axiom of enforceability: To say that detector certificates that program is secure, all executions satisfy the detector

////

We define the policies that can be enforced by an execution mechanism.

- Policies are universally quantified with respect to the execution. When we write P, we mean: for all possible sequence S, P(S) holds. This was implicit in our definition of secure (?)
- P is called the detection of the security policy.
- Policy P applies to all possible sequences in the set of execution. So a program is secure with respect to a certain policy P if for all instances S in the set of possible executions if P applied to S is true.

EM-enforceable policies

We try to understand how an enforceable mechanism behaves. Express how enforcement mechanism is characterized.

1. EM policies are universally quantified predicates over executions
   - $\forall s. P(s)$
   - **Policy P is called the detector.**
2. The detector must must be prefix-closed (with respect to sequences). This means:
   - **P(ε) holds**
   - **P(s;e) holds then P(s) holds**

- ε is a sequence. The detector must hold for the empty sequence.
- e is the last event possibly.

If we know that P(s;e) = true, then it holds for all possible subsequences

Now we fix the detector. The property P (the detector) must be prefix-closed with respect to the sequences.
This means that when I apply detector to empty sequence, detector must hold.
And if P holds for s followed by e, then P applies to s.
Those are properties we require.
We are trying to understand how security policies are enforced.

EM-enforceable policies

• Detector is the enforcer!

1. EM policies are universally quantified predicates over executions
   - $\forall s. P(s)$
   - **Policy P is called the detector.**
2. The detector must must be prefix-closed
   - **P(ε) holds**
   - **P(s;e) holds then P(s) holds**
3. If the detector is not satisfied by a sequence (the detector rejects the sequence) then it must do so in finite time
   - $\neg P(s) \implies \exists i \neg P(s[i])$

→ Property of violation: if detector is capable of finding a violation, it must do so in a finite amount of time.

If detection is not satisfied by the sequence, rejection must happen in finite amount of time.

If $\neg P(s)$ holds then there exist an index i such that NOT $\neg P(s[i])$ holds. The non satisfaction is determined in a finite amount of time.

- When is a security policy satisfied? When all possible sequences satisfy the policy.
  But this does not mean that all security policies are satisfied.
  We are just saying that in order to satisfy a policy P, all the executions must satisfy the property. We have to check this.
  First axiom of enforceability says that in order to say that the detection is a certificate that program is secure with respect to it, all the possible runs have to satisfy the property.

EM-enforceable policies

1. EM policies are universally quantified predicates over executions
2. The detector must must be prefix-closed
3. If the detector is not satisfied by a sequence (the detector recjects the sequence) then it must do so in finite time
4. **Fact**
   - **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!**

↳ property

Safety properties: Nothing bad ever happens

Safety property can be enforced **using only traces** of program

- If P(t) does not hold, then all extensions of t are also bad

Amenable to **run-time enforcement**: don't need to know future

Examples: of safety properties: can be enforced by an EM satisfying 1, 2, 3.

- **access control** (e.g. checking file permissions on file open)
- **memory safety** (process does not read/write outside its own
- memory space)
- **type safety** (data accessed in accordance with type)

Those are safety properties, enforced by execution monitor.

Liveness properties: Something good eventually happens

Not able to know what if in t1 you don't have this, you won't be closed in t2.

Nontermination: The email server will not stop running

- Violated by **denial of service attacks**

Liveness properties: Cannnot be enforced purely at run time (Something good happens in future)

Interesting properties often involve both safety and liveness

- Every property is the intersection of a safety property and a liveness property [Alpern & Schneider]

There are properties that are characterised in terms of what happens in the future.

Preventing dos is a policy in which you are not able to know if you reject a execution in finite amount of time, because if at a certain point of execution there is no DoS, this does not mean that in the future there won't be.

I'm not able to say that If at a point the server is secure it will be in

is it true to say that a policy related to liveness properties is not enforceable by an Evaluation Mechanism?

Yes, it is generally true that **liveness properties are not enforceable by an Evaluation Mechanism**, at least in the same way that safety properties are. This is because of the fundamental difference between the two:

- **Safety properties** can be enforced by an Evaluation Mechanism because they define **bad states that should never happen**. A monitor or access control mechanism can detect and prevent violations in real time. For example, a firewall can block unauthorized access attempts, ensuring that "only authorized users can access the system."
- **Liveness properties**, on the other hand, require that **something good eventually happens**, but an Evaluation Mechanism cannot always guarantee this. Just because something hasn't happened yet doesn't mean it never will. Enforcement would require an oracle-like mechanism that predicts the future, which is not feasible in practice.

You would need infinite time to understand if something good happens.
On the other hand you have SOMETHING BAD NEVER HAPPENS.

E M
E N F O R C E M E N T

EM Summary

- Analyzes the single (current) execution.

$$P(\Pi): \quad (\forall \sigma \in \Pi: \; p(\sigma))$$

- Must truncate execution as soon as prefix violates policy: $\neg p(\tau) \Rightarrow (\forall \sigma : \neg p(\tau\sigma))$

succession

→ Prefix closed

- Must detect violations after a finite time:

$$\neg p(\sigma) \Rightarrow (\exists i : \neg p(\sigma[..i]))$$

→ detection in finite amount of time

Enforceable policy implies safety property

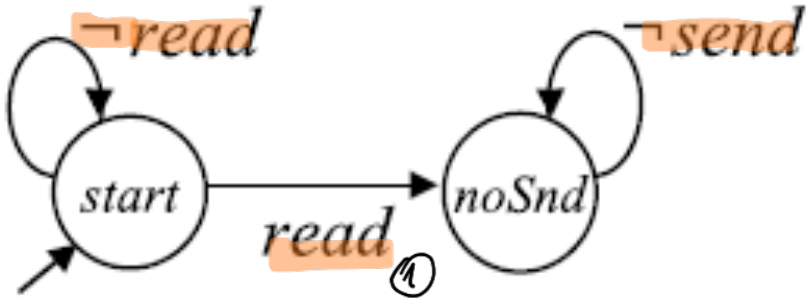
Security Automata

How do we formally represent policies?
We use special classes of Automata that represent security policies

- Security Automata (Erlingsson & Schneider 1999)
- Formalization of security policies
  - finite state automaton
  - accepts language of permissible executions (accepts sequence of executions satisfying property)
  - alphabet = set of events
  - edge labels = event predicates a property
  - all states accepting (language is prefix-closed)

if I accept one sequence I must accept all subsequences. All states are accepting

Security Automata (Example)



NO SENDS AFTER READS Confidentiality property

2 states: all states are accepting states. If I do something without read I stay in the initial state.

Transition is a predicate of the event: not read!

① Read characterizes the security policy we define. After read, any op is ok but send.

Security Automata (formally)

Finite state automata

Transition relation: EVOLUTION FROM ONE STATE TO OTHER

$$(\sigma, q) \xrightarrow{\tau}_A (\sigma', q')$$

sequence of events

TRANSITION FUNCTION

if $\sigma = a; \sigma'$
and $\delta(a, q) = q'$

event current state

otherwise

$$(\sigma, q) \xrightarrow{a}_A (\sigma', q') \qquad \text{(A-Step)}$$

A step is a move from one stage to another, and a is the event that we have.

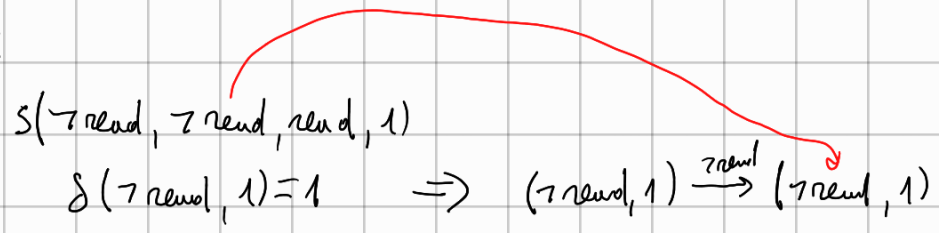
$$(\sigma, q) \xrightarrow{\cdot}_A (\cdot, q) \qquad \text{(A-Stop)}$$

Transition relation takes an event (state) and produces a new

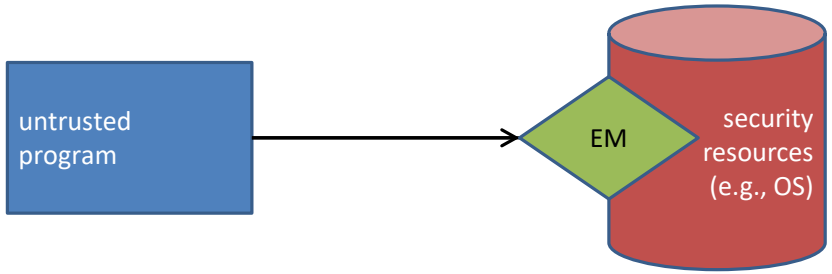
$a;b;c;d \Rightarrow$ all possible events.

$a;\sigma$ where $\sigma = b;c;d$

EX:

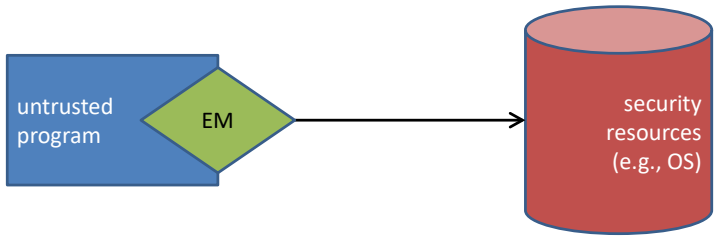


Discussion



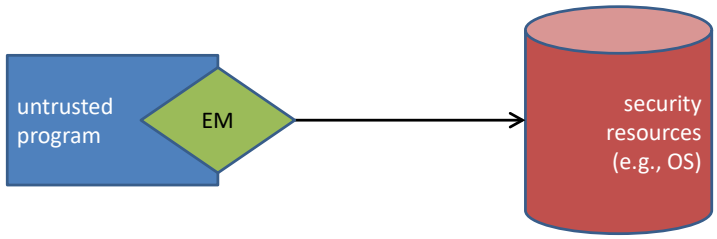
- Disadvantages of traditional (OS) EMs
  - inefficient: context-switch on every event
- Large TCB: EM extends the OS
- weak: EM can't easily see internal program actions
- non-modular: changing policy requires changing OS

In-lined reference monitor



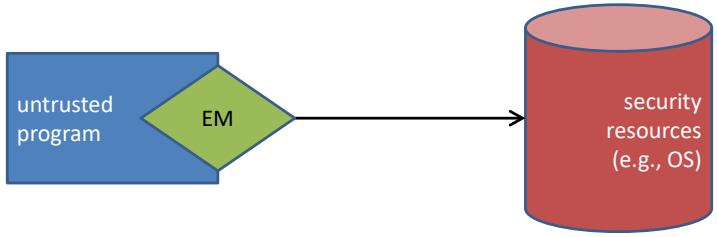
- Main idea: implement a execution monitor by *in-lining* its logic into the untrusted code:
  - In-lining procedure should be automated (by suitable compiler)

In-lined reference monitor



- Main idea: implement a execution monitor by *in-lining* its logic into the untrusted code:
  - In-lining procedure should be automated
  - The compiler automatically generate the EM code to instrument the original program

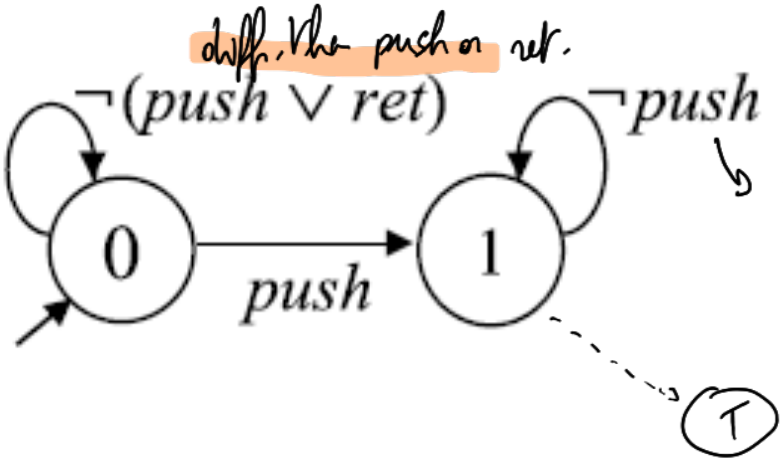
In-lined reference monitor



- Main idea: implement a execution monitor by *in-lining* its logic into the untrusted code:
- Challenges:
  - How to automatically generate EM code?
  - How to preserve (non-violating) program logic?
  - How to prevent (malicious) programs from corrupting the EM?

We have a policy as a security automata. We want a compilation strategy to generate code such that the monitor checking the property is code generated by compiler. Compiler instruments the code to define the monitor. Introduce policy as argument of compiler, not program. Code of monitor is in-lined in the program.

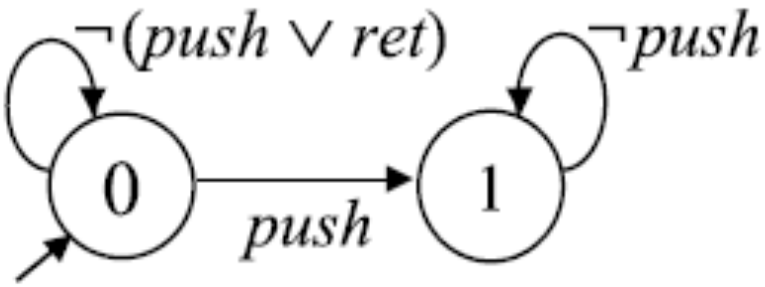
In-lining a security automaton



Policy: push exactly once before returning

In-lining a security automaton

Working not at the level of source code, but at level of code generated by compiler.



In-lining the policy in this binary code!!

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

Assembly code.

The 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 → Look at specific operation and substitute it in the automata and check if you can simplify it.
   - some edges disappear entirely
3. Generate guard code for the remaining automaton logic

Put in front of every instruction (any instruction is characterized by a guard of the automata)

In-lining: the actual steps

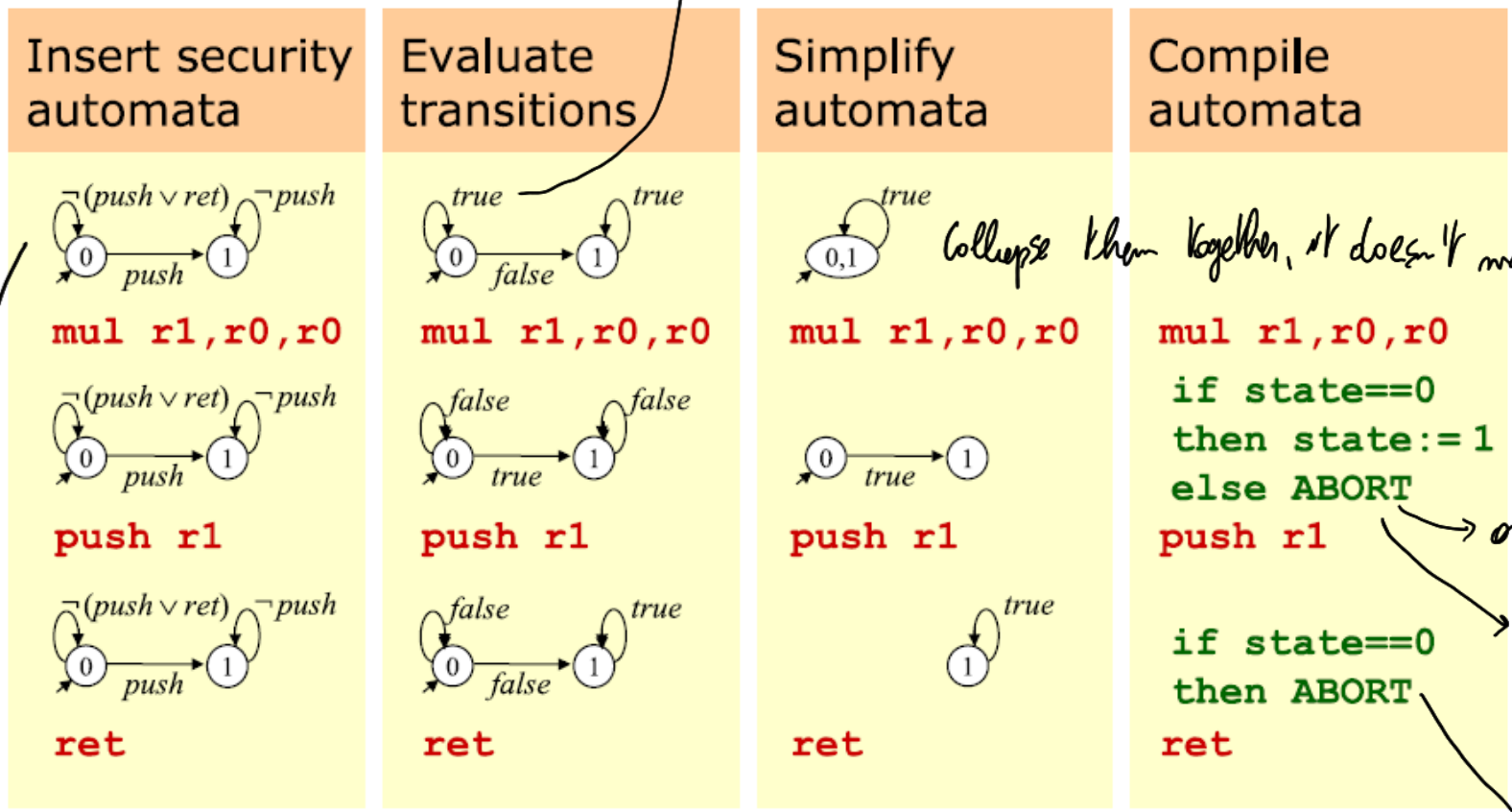
Insert security automata.
Inserts a copy of the security automaton before each target instruction.

Evaluate transitions.
Evaluates any transition predicates that can be given the target instruction.

Simplify automata.
Deletes transitions labelled by transition predicates that evaluated to false.

Compile automata.
Translates the remaining security automata into code.
FAIL is invoked by the added code if the automaton reject its input.

Example



¬(push ∨ ret) is satisfied. So true

Collapse them together, it doesn't matter what we are doing. We stay in that state

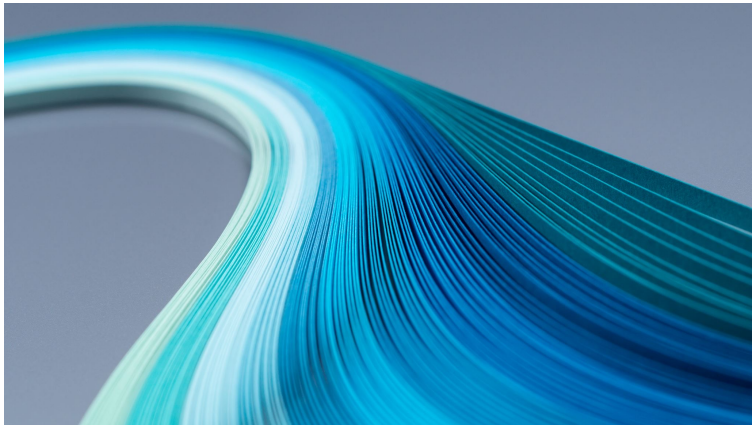
only transition we expect.

if state is 0, ok otherwise violation

if state is 0, violation

insert it in front of every operation. Each step must satisfy policy.

* We don't have to do anything.

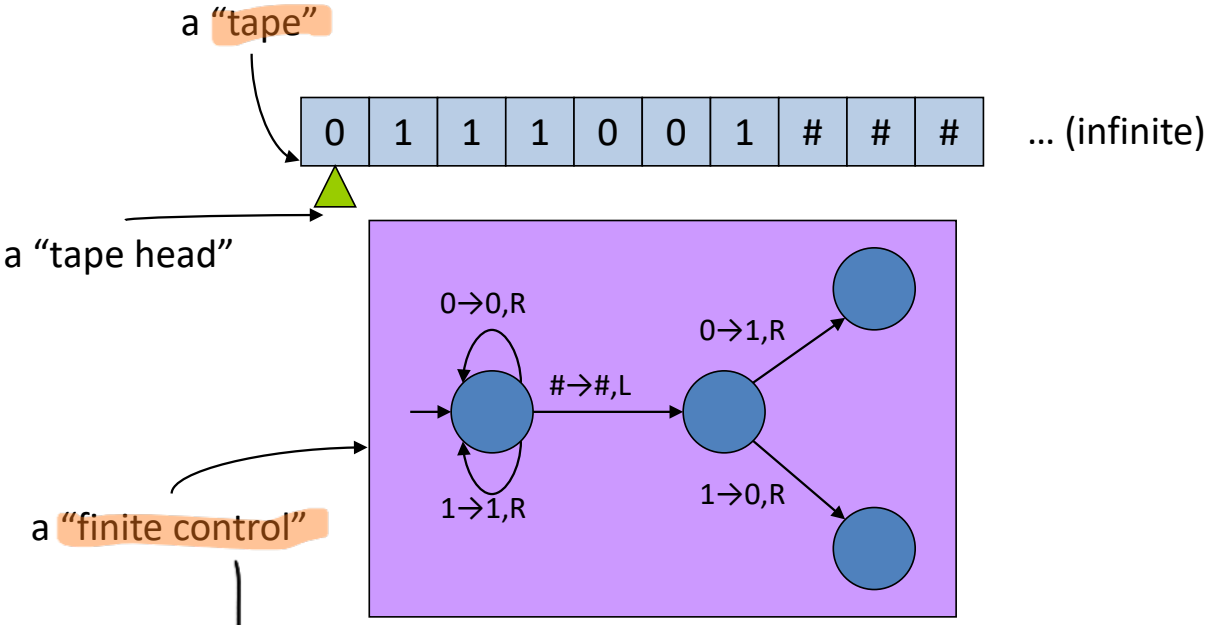


A BIT OF THEORY

WHAT EVERY HACKER SHOULD KNOW ABOUT THEORY OF COMPUTATION

Turing Machines and Computability

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



Turing machine: model of computer. Infinite tape in which you can move and modify

TM Expressive power

Can do simple arithmetic

TMs don't necessarily terminate

Can evaluate a C program encoded in binary

Can simulate C behavior

Can simulate arbitrary TMs (given as input) on arbitrary inputs (given as input): the "**universal TM**"

Fact: **Can do anything a real computer can do** (**but very, very slowly**)

TMs can't solve **undecidable problems** (e.g., **halting problem**)

A Universal Turing Machine is a special kind of Turing Machine that can simulate **any other Turing Machine** when given a description of that machine and an input for it.

In simpler terms, imagine you have a machine that can read a program (a description of another Turing Machine) and then execute that program on some input. This is similar to how modern computers work: a computer itself doesn't just perform one fixed task but can run different programs based on what you load into it.

So, the phrase means:

- The Universal Turing Machine can take as input a description of **any** Turing Machine (the "arbitrary TM").
- It can also take an **input** for that machine (the "arbitrary input").
- It then **simulates** how that machine would behave on that input.

The halting problem

- The **halting problem** is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever.
- Turing proved a general algorithm to solve the halting problem for all possible program-input pairs cannot exist.
  - **the halting problem is undecidable over Turing machines**
- Turing result is significant to practical computing efforts, **defining a class of applications which no programming invention can possibly perform perfectly.**

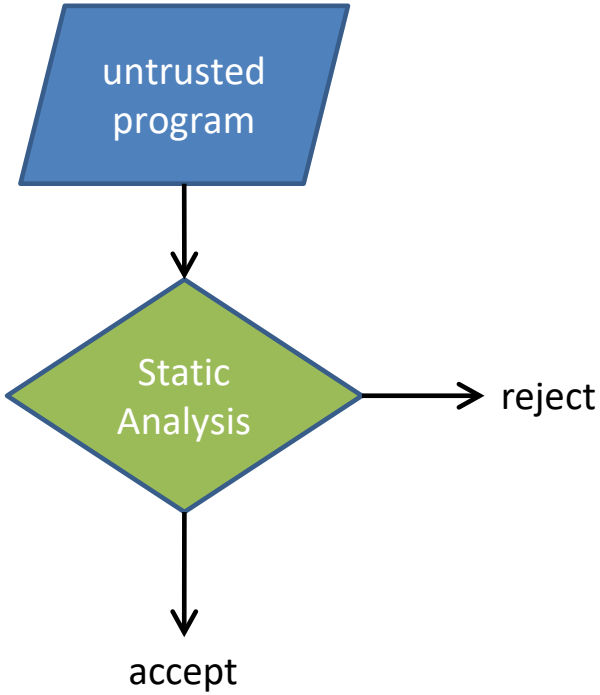
In terms of computability we can define class of RE problems. They are a decision problem: you have binary result: "yes, true", "no, false".

Computation theory

- RE (recursively enumerable) is the class of decision problems for which a 'yes' answer can be verified by a Turing machine in a finite amount of time.
  - if the answer to a problem instance is 'yes', then there is some procedure which takes finite time to determine this, and this procedure never falsely reports 'yes' when the true answer is 'no'. However, when the true answer is 'no', the procedure is not required to halt
- co-RE is the set of all languages that are complements of a language in RE.
  - co-RE contains languages of which membership can be disproved in a finite amount of time, but proving membership might take forever.

ENFORCEMENT STRATEGIES

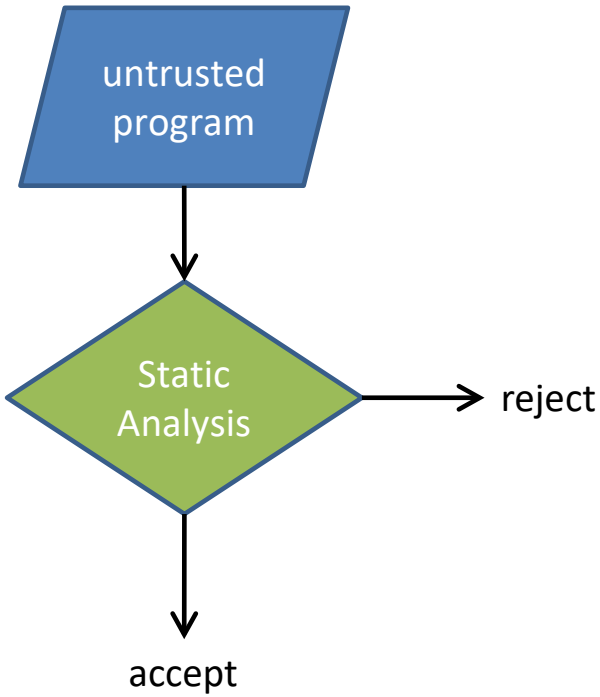
Take 1: static analysis



- 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…?

HALTS IF ACCEPT
NOT NECESSARILY HALTS IF REJECTS

Take 1: static analysis



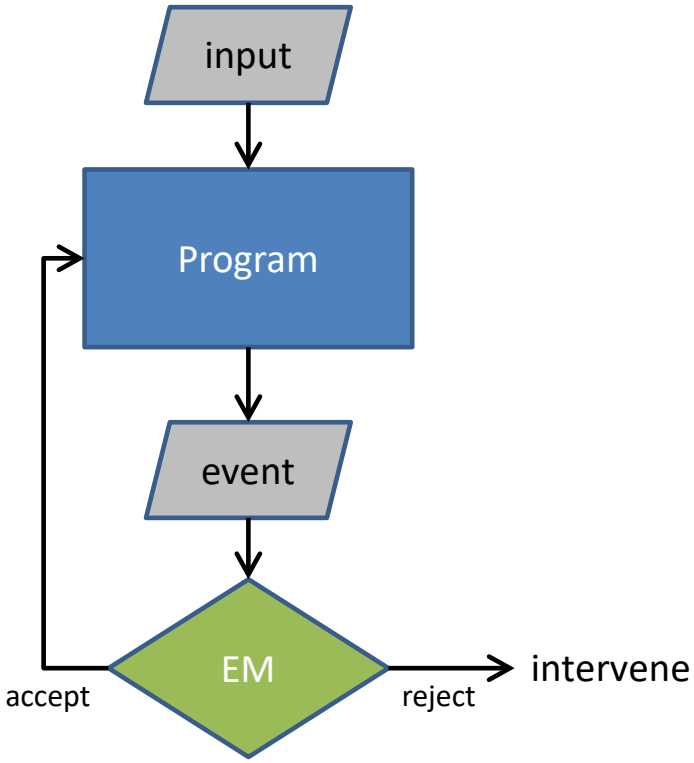
- Approach:
  - analyze untrusted code BEFORE it runs
  - return "accept" or "reject" in finite time
- Pros:
  - immediate answer
  - code runs at full speed
- Cons:
  - high load overhead
  - weak in power…?

Recursively Decidable Policies

You cannot disprove that it is not trusted

↳ You can only decide on a finite amount of time that a program is trusted

Take 2: EMs

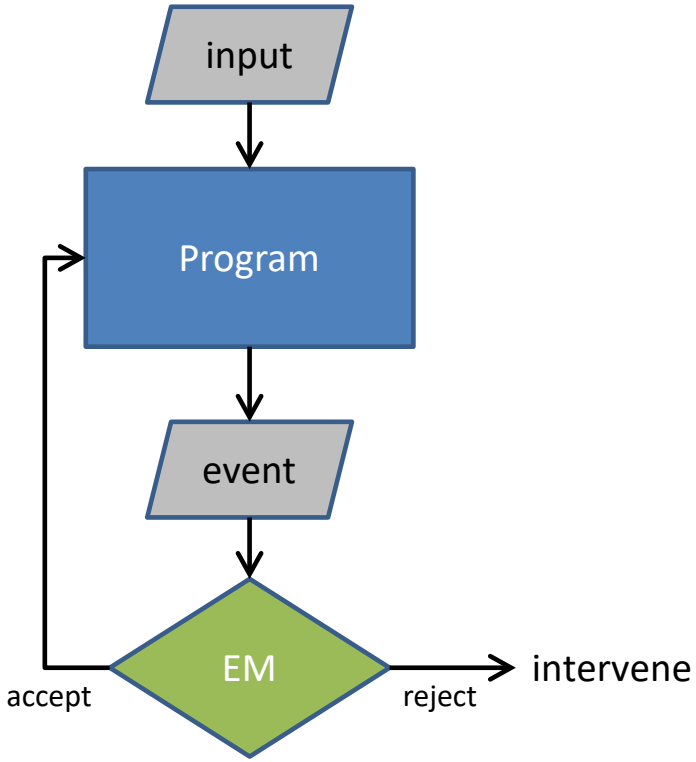


- Approach:
  - EM monitors events
  - intervenes to prevent violations
  - implemented outside program
- Cons:
  - no answer until execution
  - runtime slow-down (context-switches)
- Pros:
  - lower load-time overhead than static analysis
  - more powerful…?

HALTS IF REJECTS

NOT NECESSARILY IF IT ACCEPTS

Take 2: EMs



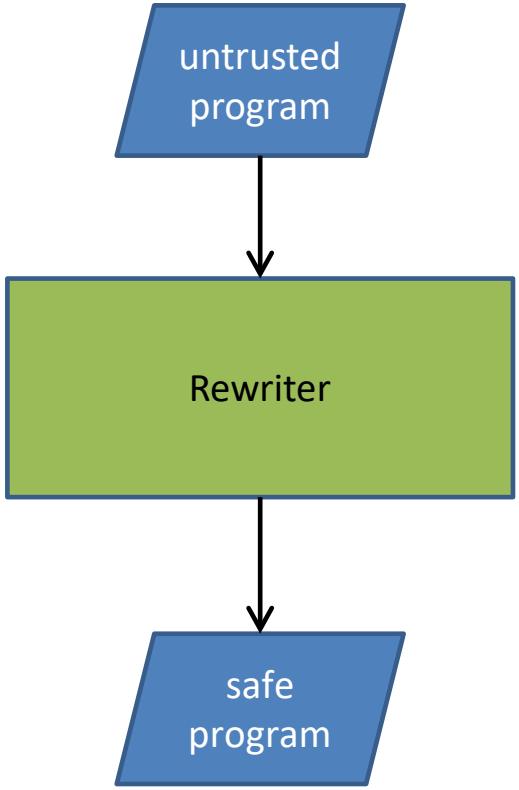
- Approach:
  - EM monitors events
  - intervenes to prevent violations
  - implemented outside program
- Cons:
  - no answer until execution
  - runtime slow-down (context-switches)
- Pros:
  - lower load-time overhead than static analysis
  - more powerful…?

co-Recursively Enumerable Policies

We are capable of deciding that a possible error occurs, but

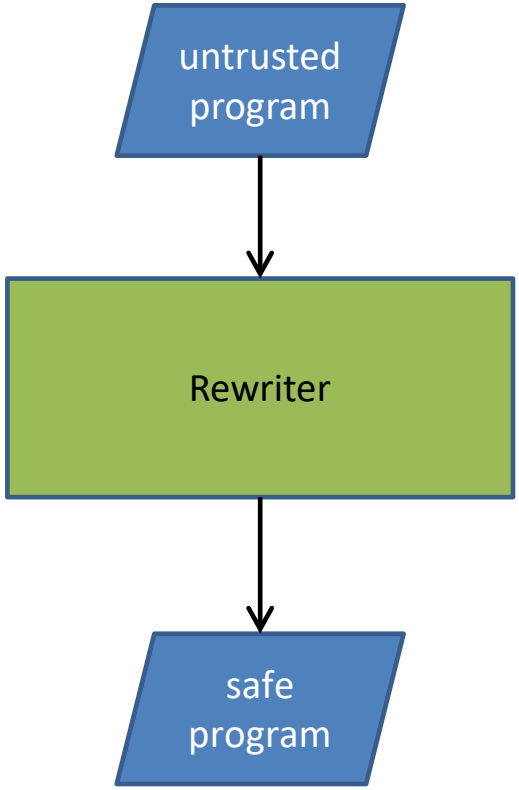
in line reference monitor: instrumentation of code

IRM Strategy



- 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

IRM Strategy



- 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

Answer is much more complicated: you have to decide with respect also to the equivalence of starting program and ending one. Translation changes. The equality of source vs target is a problem.

See Computability Classes for Enforcement Mechanisms on TEAMS

IRMs are **practically useful** for enforcing many security properties:

1. **Memory Safety** (e.g., Rust's Borrow Checker)
   - Ensures memory access follows safe ownership rules.
   - Works through **statically enforceable** constraints.
2. **Access Control** (e.g., Java Security Manager)
   - Prevents unauthorized access at runtime using inline security checks.
   - Uses **dynamic enforcement**, but restricted by predefined permissions.
3. **Control Flow Integrity (CFI)**
   - Prevents exploits by ensuring execution follows a valid control flow.
   - Requires **finite-state tracking** and lightweight IRM modifications.
4. **Taint Tracking**
   - Tracks untrusted data to prevent injection attacks.
   - Requires **context-sensitive** enforcement, which can be **computationally complex**.

READING

- FRED B. SCHNEIDER Enforceable Security Policies, ACM Transactions on Information and System Security, Vol. 3, No. 1, February 2000
- U Erlingsson, FB Schneider SASI enforcement of security policies: A retrospective Proceedings DARPA Information Security 2000
- Computability Classes for Enforcement Mechanisms
  KEVIN W. HAMLEN Cornell University GREG MORRISETT Harvard University and FRED B. SCHNEIDER Cornell University
- Available on TEAMS