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
- FAT Pointer & Data Descriptor

Static Analysis

- Monotone Framework

What we have learned (example)

- Stack canaries:
  - **Run-time support**
  - **Code instrumentation checking at run-time the compliance with the security policy**
- **Stack Canary Security policy: the control flow cannot be alterered**

A motivating example

A TRUSTED PROGRAM

:

10: z := Mem(x)

11: if(i ≥ 0) jump y

:

The program dereferences memory and makes use of indirect control flow transfer.

THE ATTACKER CODE

20: i:=0

21: x := *attacker's desired address*

22: y:=24

23: f(0 = 0) jump 10

24: *copy memory contents from* z

The attacker sets things up in a way so that unauthorized memory is copied into x

the attacker can later access

A motivating example

A TRUSTED PROGRAM

:

10: z := Mem(x)

11: if(i ≥ 0) jump y

:

THE ATTACKER CODE

20: i:=0

21: x := ***attacker's desired address***

22: y:=24

23: f(0 = 0) jump 10

24: ***copy memory contents from z***

Form of return-oriented programming (ROP): the attacker knows program text.

This relies on the ability to change control flow using indirection so that commands are executed in the order needed by the attackers to carry out their goals.

Countermeasure: sandbox

We (aka the compiler) instruments the original program with "code sandbox" to manage flow control between commands at program point l and program point h (i.e. pc_l and pc_h).

The instrumentation

The compiler rewrites indirect jump commands to ensure that their target always lies within the right bounds (l, h).

$$\textit{Rewrite all } \ \texttt{if}(Q)\,\texttt{jump}\,e \ \textit{ commands as } \ \texttt{if}(Q)\,\texttt{jump}\,(e\,\&\,\texttt{pc}_h) \mid \texttt{pc}_l$$

Our approach

Enforce a general safety property that takes aspects of control flow and program state into account.

Example, suppose that our programming language has the ability to make three types of function calls,

send and receive from the network

read from a local file.

Then we want to enforce a safety policy on untrusted code which says that send cannot be called after read.

Next step: security policies

Introduce suitable mechanisms for

1) Defining security policies

2) Implementing enforcement mechanisms via compiler instrumentation

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? • Are some enforcement approaches strictly more powerful than others?	• what is the landscape of policies, policy classes, and enforcement mechanisms?

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.

Execution Monitor

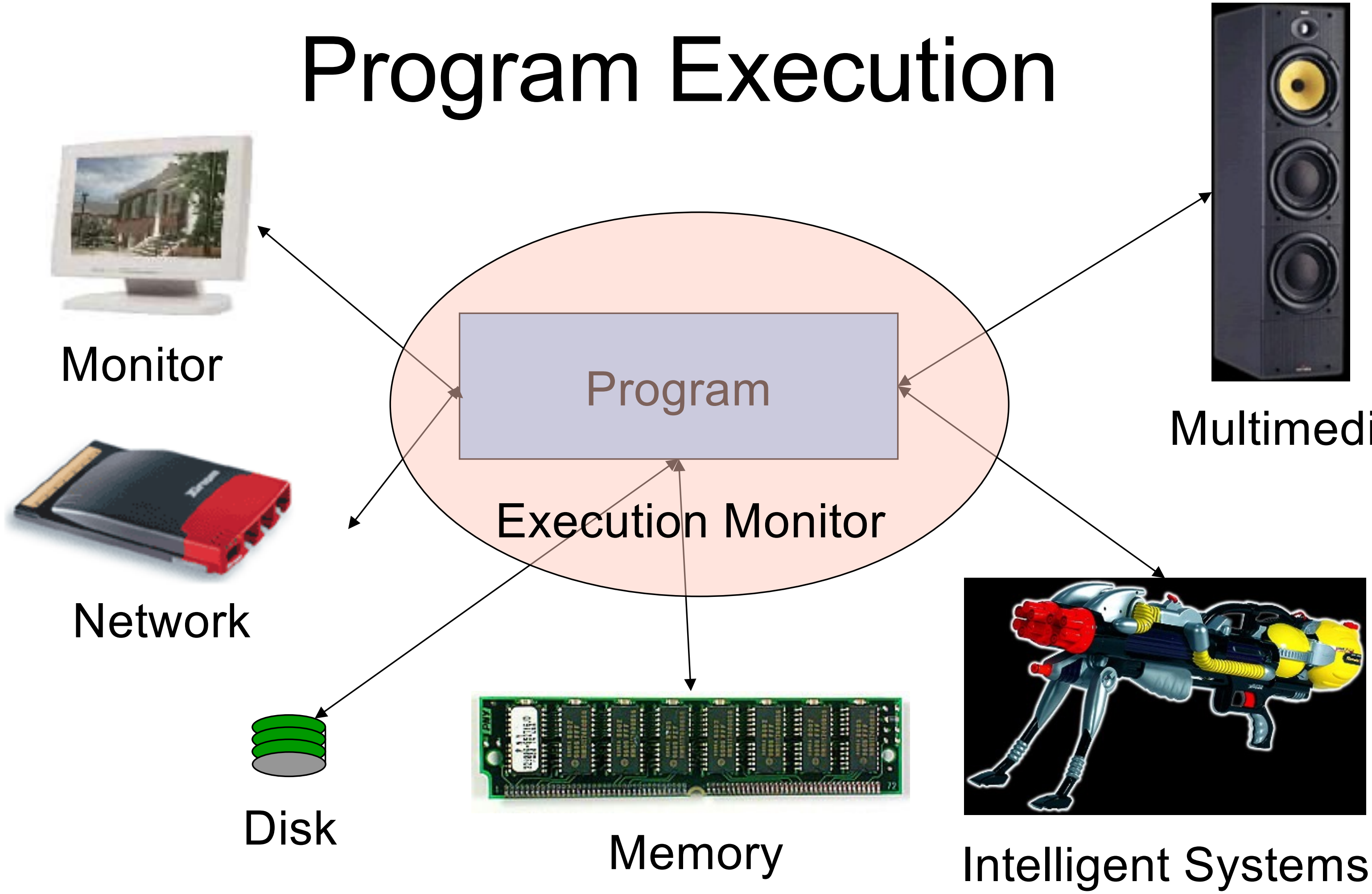
Execution Monitors (EMs)

- 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)

The Context: Program Execution



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 build this?

Probably not …....

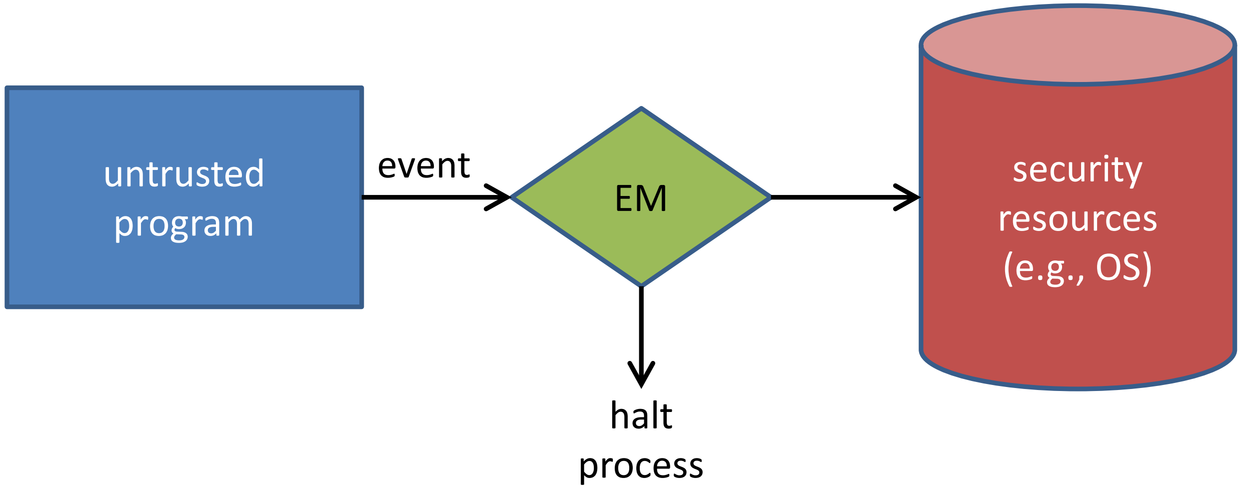
~~Ideal~~ Real Execution Monitor

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

Operating Systems

- Provide execution monitors for most security-critical resources
  - When a program opens a file in Unix or Windows, the OS checks that the principal running the program can open that file
- Doesn't allow different policies for different programs
- No flexibility over what is monitored
  - OS decides for everyone
  - Hence, can't monitor inexpensive operations

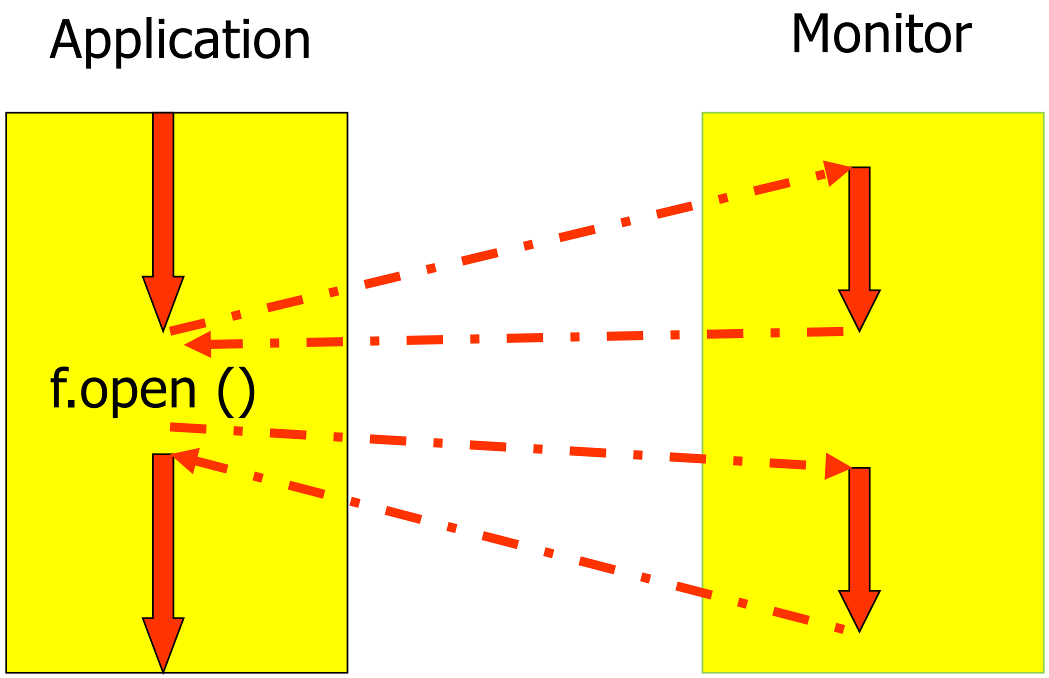
OS Execution Monitor



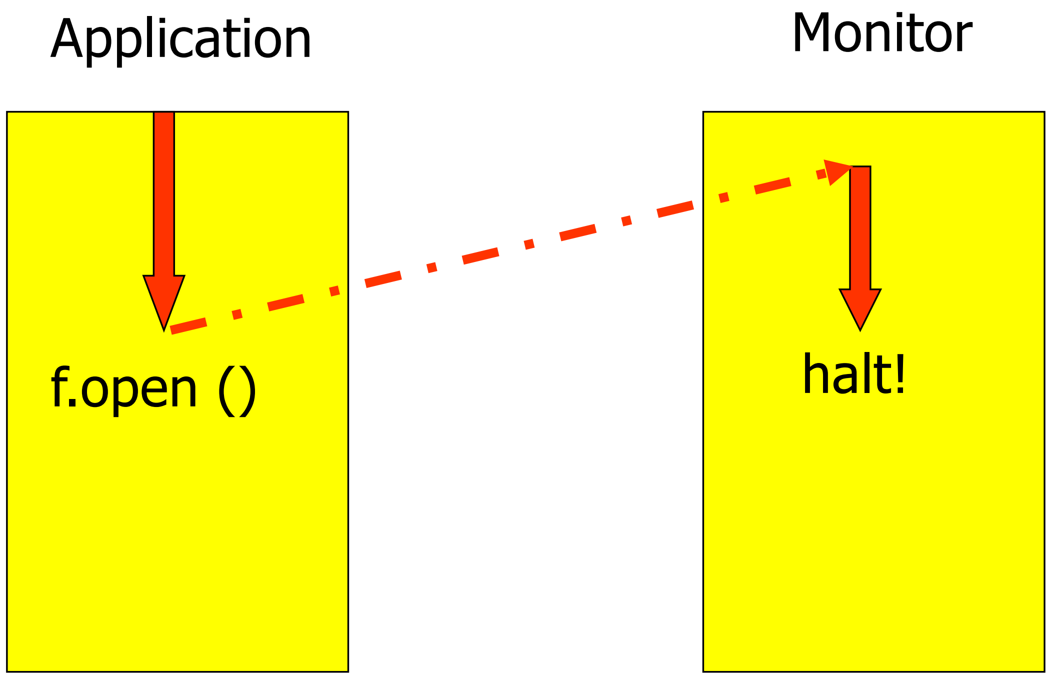
Execution Monitor

- EMs are run-time modules that runs in parallel with an application
  - monitors may detect, prevent, and recover from application errors at run time
  - monitor decisions may be based on **execution history**

EM: Good Operations

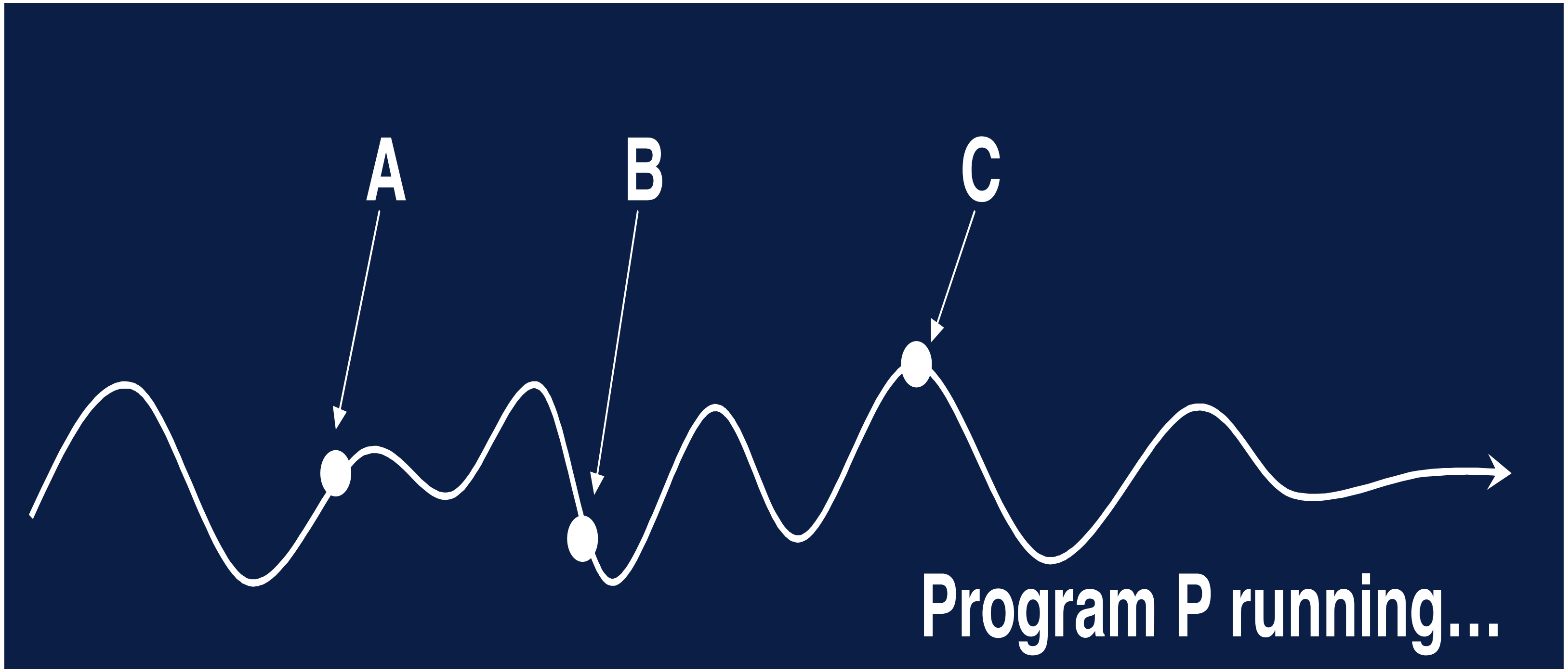


EM: Bad Operation



Intuition

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



What policies can be enforced?

- Assume:
  - Security Automaton can see entire state of world, everything about instruction about to execute
  - Security Automaton has unlimited memory, can do unlimited computation
- Are there interesting policies that still can't be enforced?

What's a Security Policy?

- What's a program?
  - 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

Programs and Policies

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

Programs and Policies

- A program **S** is a **set** of sequences (possible executions)
  - A program is modelled as the set S = $\{s_1, s_2, \ldots\ldots\}$
- A policy **P** is a **property** of programs
- A policy partitions the program space into two groups:
  - Permissible
  - Impermissible
- Impermissible programs are censored somehow(e.g.,terminated on violating runs)

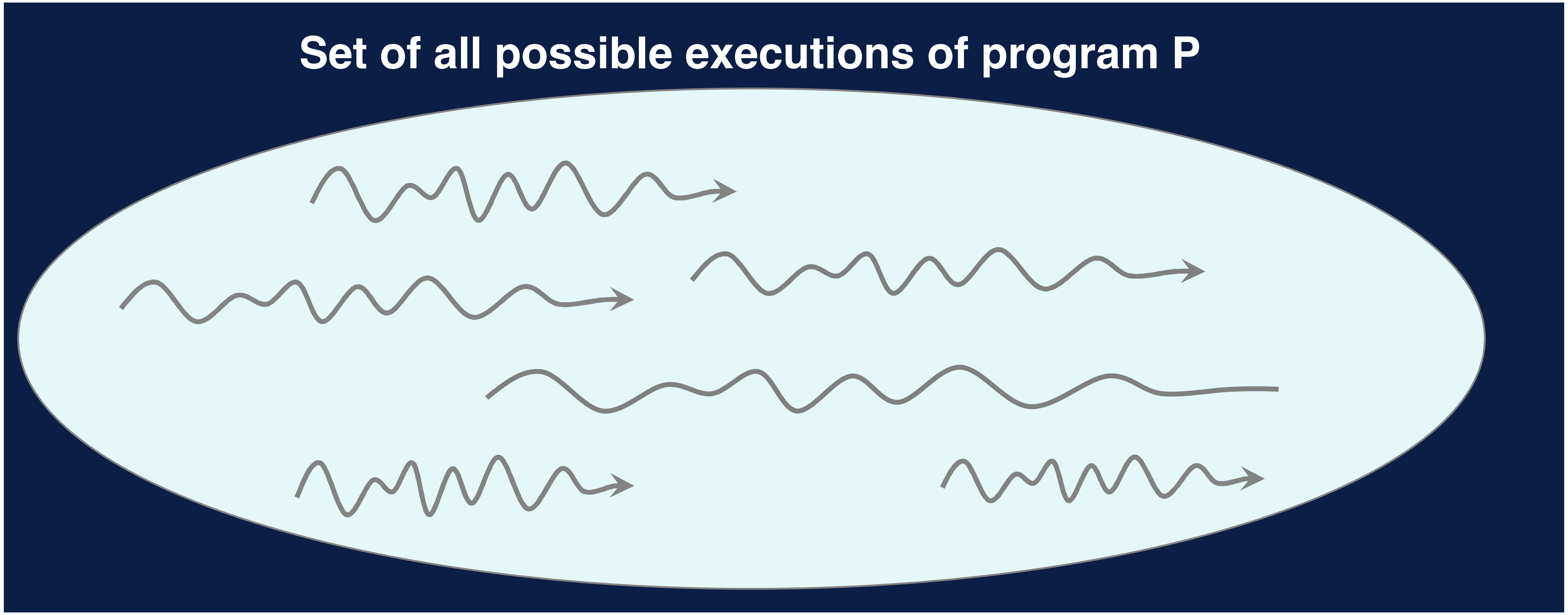
Formally…

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

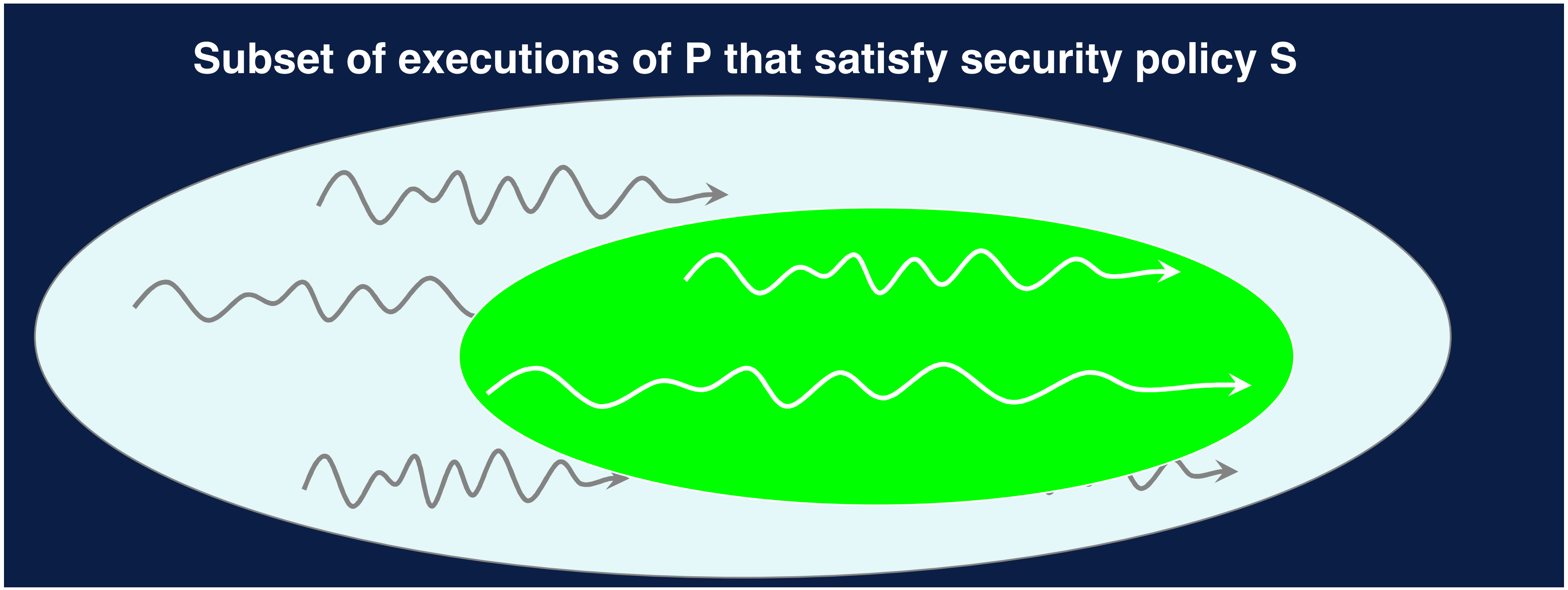
  set of executions $\rightarrow$ Boolean

S is safe iff $P\ (\Sigma_S)$ is true.

Execution Traces



Security Policies



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.
- **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