# Run-time Enforcement of Policies

Harshit Shah
Tata Institute of Fundamental Research
and Uni of Tento
R.K Shyamasundar
IBM India Research Lab

(Under Indo-Italian Programme (TIFR-Trento)- ITPAR)

# Organization

- Motivation
- Policy Specification
  - Guarded Specifications
  - Past temporal Logic
- Implementation
- Comparison
- Summary

# Untrusted code

- **Open a backdoor**

  Ken Thomson - Reflections on trusting trust

  Malicious code in the compiler that opens a backdoor
- **Leak sensitive data**

  Keylogger.Stawin: attempts to steal user's online banking information
- **Corrupt the system**

  CIH (Chernobyl) virus: over-writes critical system information and corrupts BIOS
- **Infect other hosts on the network**

  Mydoom: fast spreading worm that sends junk e-mail through infected computers

# Approach: Monitoring & Enforcing Policies

1. **Monitor activity of untrusted code**

   observe what the code is trying to do

2. **Formulate policies to constrain activity**

   state what constitutes “undesirable” behaviour

3. **Enforce the policy on untrusted code**

   ensure that program actions are consistent with definition of safety in step 2.

# Monitor the activity

- Observable program actions
  - System calls e.g., `open(``abc.txt'',O_RDONLY)`
- Applications access OS services through system call
  - e.g., using a network interface card, creating a file

# IBM Tivoli – Monitor, Alert and Correct

- keep an eye on all the parts of your infrastructure and alert the right people, or even take corrective actions, when things go wrong.
  - OS agents monitor general system resources,
  - while application agents monitor resources specific to that application.
  - ….

# Policy Specifications : OS Monitoring

- Policy of Access
- Default Policy
  - For those that have not been specified (like Symantic firewall/security agent)
- Privilege elevation
  - Beyond restricting an application to its expected behavior, there are situations in which there is a need to increase its privilege.

# Privilege Elevation; Examples

- Unix -- many system services and applications require root privilege to operate.
  - Often, higher privilege required only for a few operations.
  - Instead of running the entire application with special privilege, elevate the privilege of a single system call.
  - the principle of least privilege: every program and every user should operate using the least amount of privilege necessary to complete the job
- specifying the requirement that certain actions require elevated privilege, the policy language needs to assign the desired privilege to matching policy statements.
  - start the program in the process context of a less privileged user and the kernel raises the privilege just before the specified system call is executed and lowers directly afterwards.
  - Restrictions on user daemon and the system daemon

# Privilege Elevation; Examples

- Identifying the privileged operations of setuid or setgid applications allows us to create policies that elevate privileges of those operations without the need to run the whole application at an elevated privilege level.
- As a result, an adversary who manages to seize control of a vulnerable application receives only very limited additional capabilities instead of full privileges.

# Examples ..

- Ping program-- a setuid application requiring special privileges to operate correctly.
  - To send and receive ICMP packets, ping creates a raw socket which is a privileged operation in Unix.
  - With privilege elevation, we execute ping without special privileges and use a policy that contains a statement granting ping the privilege to create a raw socket.

# Examples …

- Unix allows an application to discard privileges by changing the uid and gid of a process.
  - The change is permanent and the process cannot recover those privileges later.
- If an application occasionally needs special privileges throughout its lifetime dropping privileges is not an option.
  - privilege elevation becomes especially useful.
  - E.g., , the ntpd daemon synchronizes the system clock. Changing system time is a privileged operation and ntpd retains root privileges for its whole lifetime.

# Our Objective

- A Simple language for specifying policies (including OS monitoring …)
- Generating monitors from such a specification
- Verify properties of the various policies being enforced
- Later use past LTL for specifying policies for temporal requirements and distribute
  - Have the power of shallow automata (where the order of access does not matter)
  - Security monitoring automata, edit automata ..

# Formulation of policies

- Guarded Command: *G -> S*
  - If proposition *G* is true, then execute statements in *S*
- Guarded Comand Policy Specification Language (GCPSL)
- Verifiability of policies (consistencey etc)

# GCPSL syntax

**state:**

$var\_type_1 \quad state\_var_1 = initial\_value_1$;

……

**command:**

$(call_1(x,y,z)) \wedge (cond_1 \vee cond_2)$ **->** $statement_1$; …;

$(call_2(w)) \wedge (cond_3)$ **->** `terminate`;

… …

**default:**

`skip` **|** `terminate;`

# GCPSL example

**Application should not write more than 80,000 bytes**

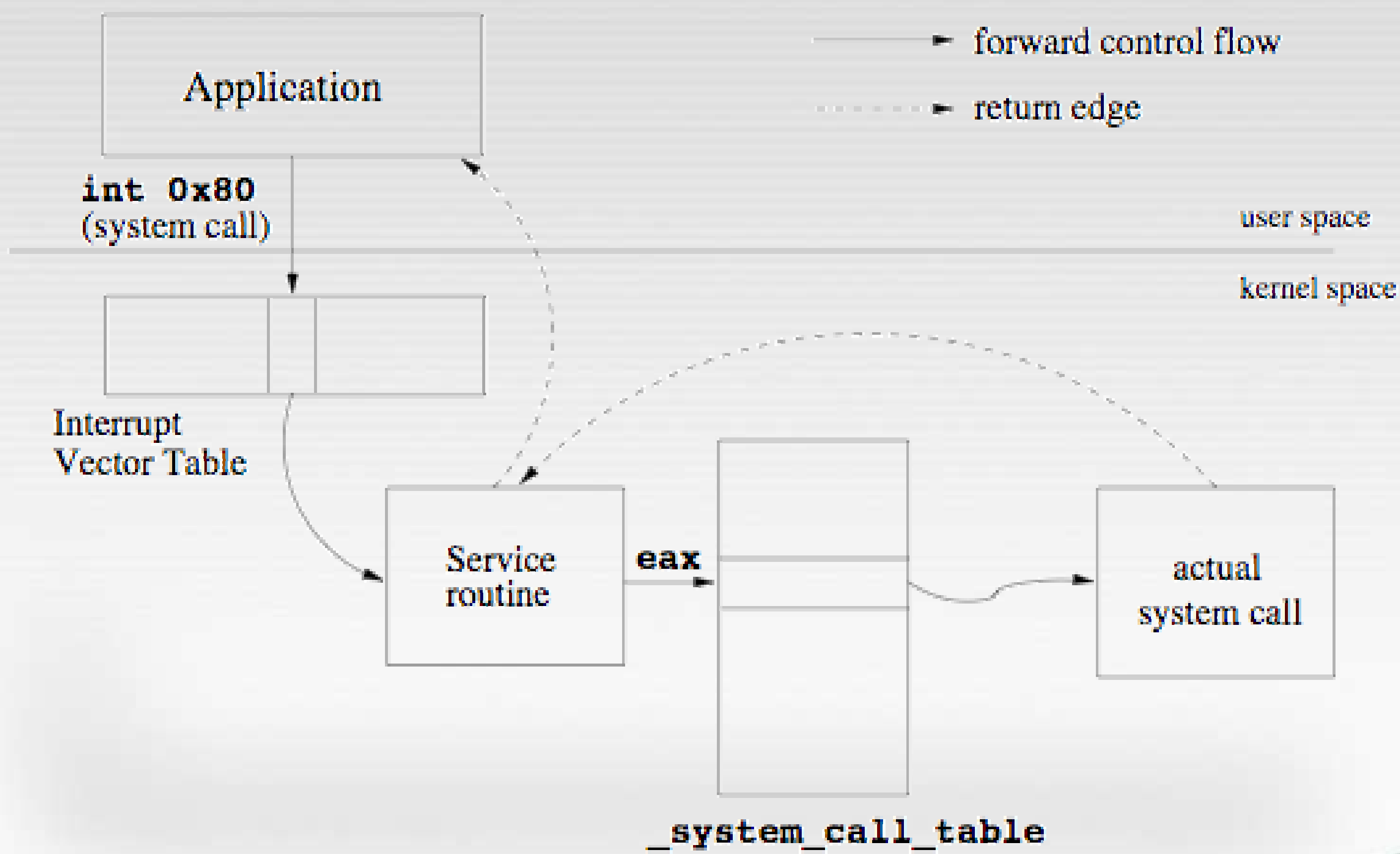
```
state:
    int    count = 0;
command:
    (write(fd, buff, num_bytes)) /\ (count < 80000) /\
    (count + num_bytes    80000) ->
            count = count + result;

    !(write(fd, buff, num_bytes) ->
            skip;
default:
    terminate;
```

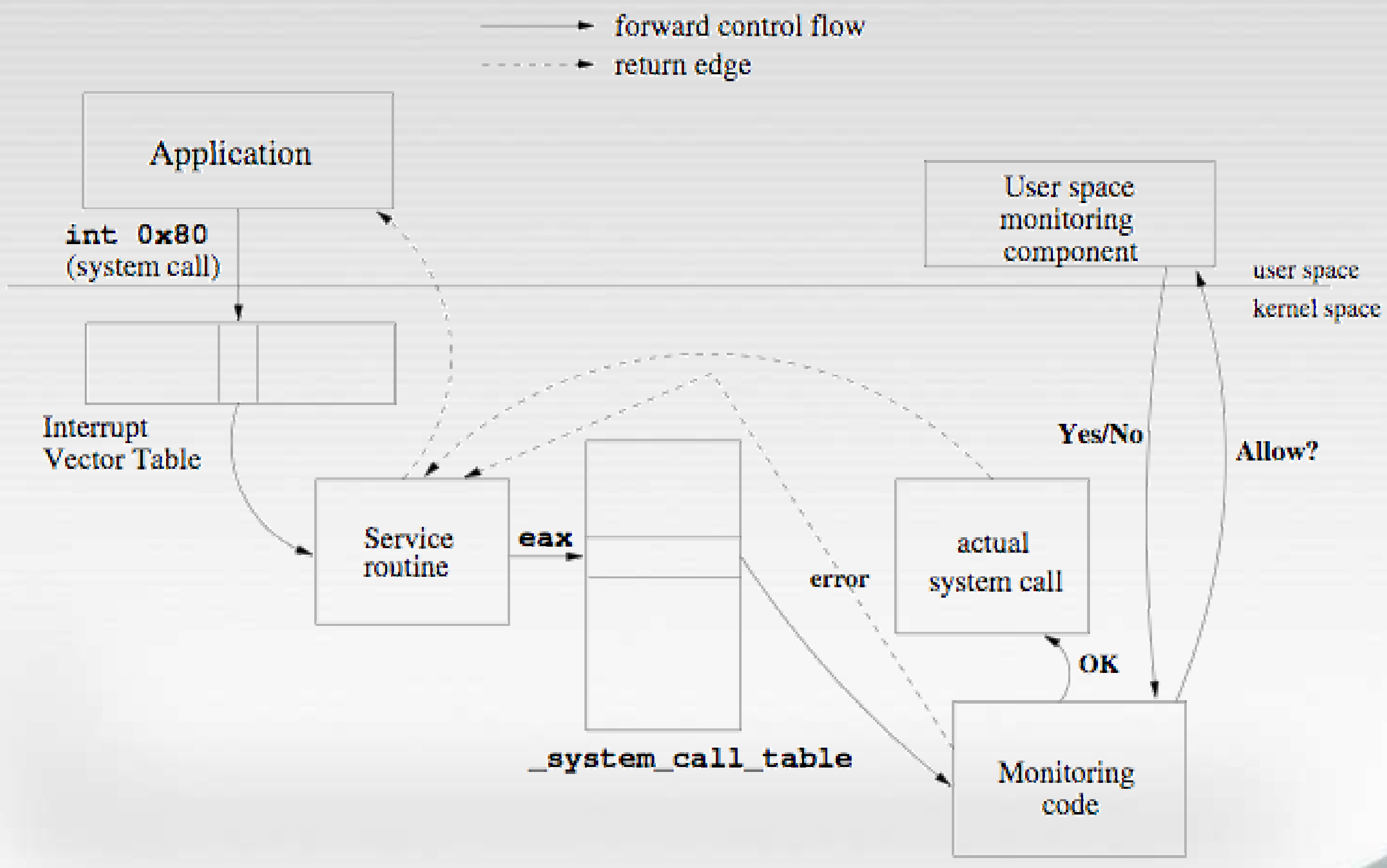
# Enforce policy

- Intercept system calls
- Consult policy
- Allow / disallow system call
- Check consistency of policies or others safety properties  (using a UNITY like methodology)

# System call architecture



# Monitoring framework

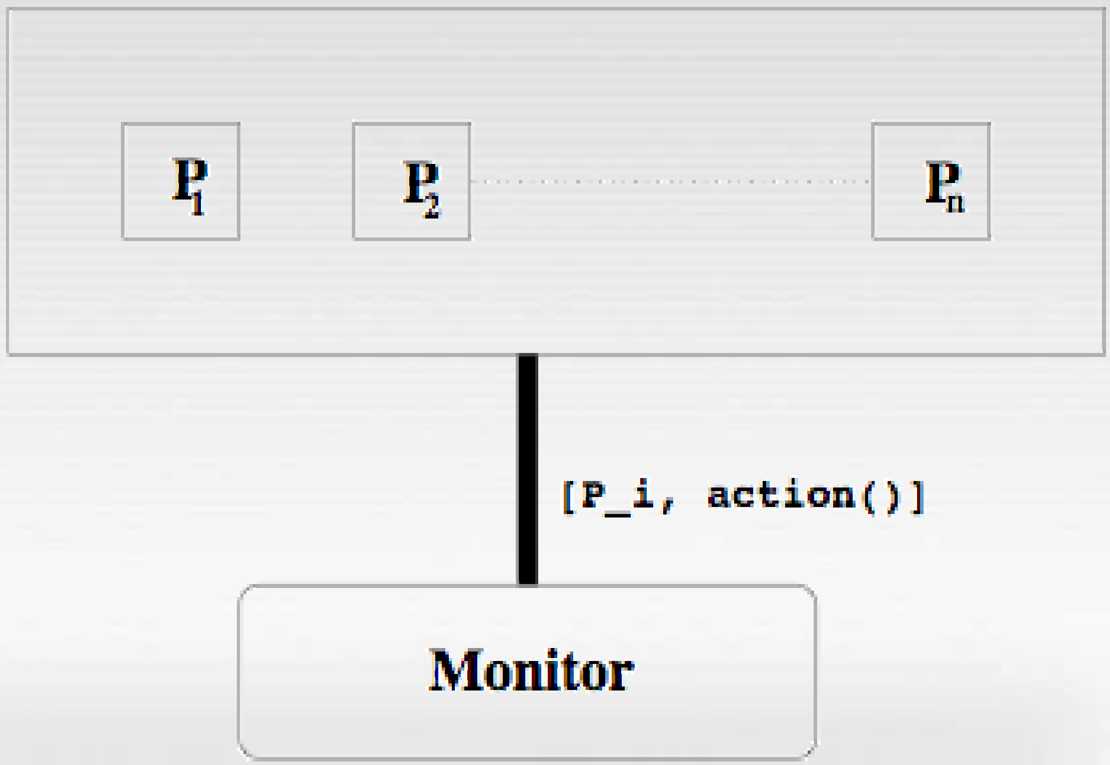


# Monitoring a set of processes

- E.g., all sessions of an application can send N messages (collectively) after which each session should acquire user permission before sending a message

- Global state variables for the aggregate behaviour
- A copy of local variables for each process
- Visible actions of the form <$P_i$,call()>

# Monitoring a set of processes



# Enforcing temporal constraints

- Write the policy as a Pure Past LTL formula
- Compile into an automaton
- Use this automaton as monitor specification

# Past LTL -- Operators

$\varphi :\, = p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi\, \mathcal{S}\, \varphi \mid \mathcal{Y}\varphi$

Here, $p \in AP$ is an atomic propostiion, $\mathcal{S}$ is the "since" operator and $\mathcal{Y}$ is the "yesterday" or the "previous step" operator. Other operators can be expressed in terms of these as follows:

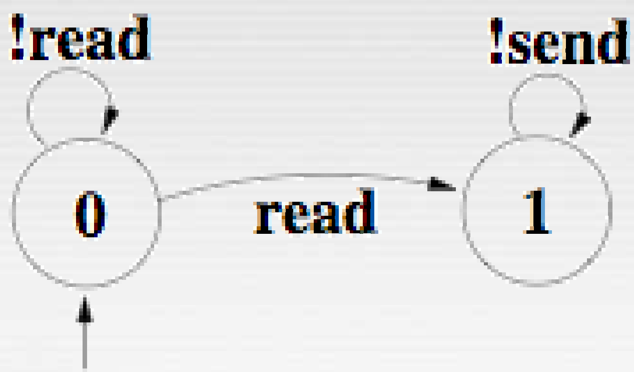
| | |
|---|---|
| $\mathcal{O}\varphi \equiv \mathbf{T}\, \mathcal{S}\, \varphi$ | ($\mathcal{O}$ - once in the past) |
| $\mathcal{H}\varphi \equiv \neg\mathcal{O}\neg\varphi$ | ($\mathcal{H}$ - always in the past) |
| $\overline{\mathcal{Y}}\varphi \equiv \neg\mathcal{Y}\neg\varphi$ | (weak version of $\mathcal{Y}$) |
| $\varphi\overline{\mathcal{S}}\psi \equiv \mathcal{H}\varphi \;\vee\; \varphi\mathcal{S}\psi$ | (weak version of $\mathcal{S}$) |

# Past LTL

- Synthesize into Security Automata
  - Monitoring, Edit automata
  - Shallow automata
- Also can be expressed as an Lustre (reactive program)

# Example

- **H(*send*->!O(*read*))** "No message can be sent after a file is read"




Security Automaton

**state**:
int $q$=0;
**command**:
($q$==0)/\(!read())->$q$=0;
($q$==0)/\(read())->$q$=1;
($q$==1)/\(!send())->$q$=1;
**default:**
**terminate**

# Sample Implementation

- Linux kernel 2.6.17
- Policy: process cannot write more than 550bytes
- The monitor forks and calls the process
- Kernel intercepts “write” calls by untrusted process and signals the monitoring process
- Monitoring process writes the response into a file read by kernel module
- Normal Performance
- Performance from external Monitor:
  - including calling the monitor, initialization, fork and running the process,communicating with kernel module, etc.: High
- Total time taken when the state information is maintained inside kernel (no monitoring process in user space): Quite Efficient
- Switches between kernel and user space for every system call incur extra time

# Comparisons

# Other tools: systcalltracking

- Logs system wide calls based on rules

```
rule {
    syscall_name = open
    when = before
    action { type = LOG }}
```

- Static patterns
- Cannot enforce fine-grained policies on individual processes
- Cannot enforce temporal constraints

http://syscalltrack.sourceforge.net/

# Systrace

- Policy specifies action for each system call
- Interactive policy generation
- GUI
- Remote monitoring
- Lengthy policies

http://www.citi.umich.edu/u/provos/systrace/

# Systrace policy

```
Policy: /bin/ls, Emulation:
native
native-munmap: permit
[...]
native-stat: permit
native-fsread: filename match "/usr/*" then permit
native-fsread: filename eq "/tmp" then permit
native-fsread: filename eq "/etc" then deny[enotdir]
native-fchdir: permit
native-fstat: permit
native-fcntl: permit
[...]
native-close: permit
native-write: permit
native-exit: permit
```

# Other Frameworks

- Naccio
  - Implemantation for windows
  - Writing policies is complex
- PoET/PSLang
  - Monitoring for Java
- Polymer
  - Edit automaton implementation for Java programs

# Overall comparison

| | Syscalltrac king | Systrace | Naccio | PoET/ PSLang | Polymer | GCPSL |
|---|---|---|---|---|---|---|
| Ease of writing policies | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ |
| Expressive policies? | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ |
| constrain collective behaviour of a set? | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| Platforn | Linux | Linux | Win32/Jav a | Java | Java | Linux |
| Target code modified? | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ |

# Summary

- GCPSL allows a rich set of policies to be formulated
- Policies are easy to write
- Verifiability of policies
- The target program does not need to be modified
- More experiments for fine-grained specifications and temporal specifications need to be done
- Integration …

# Thank You

# Sample Implementation

- Linux kernel 2.6.17
- Policy: process cannot write more than 550bytes
- The monitor forks and calls the process
- Kernel intercepts "write" calls by untrusted process and signals the monitoring process
- Monitoring process writes the response into a file read by kernel module
- Total time taken for the original process: 2ms
- Total time with monitoring (including calling the monitor, initialization, fork and running the process,communicating with kernel module, etc.): 190ms
- Total time taken when the state information is maintained inside kernel (no monitoring process in user space): 3.1ms
- Switches between kernel and user space for every system call incur extra time