A Sandbox with Dynamic Policy Based on Execution Contexts of Applications

Tomohiro Shioya, Yoshihiro Oyama, Hideya Iwasaki

The University of Electro-Communications, Japan

Outline

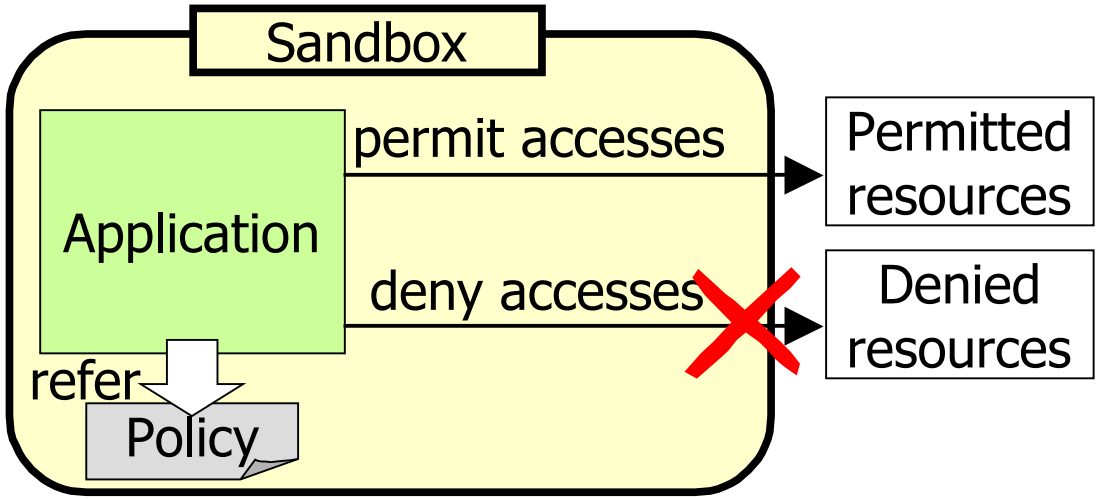
1. **Background and Motivation**
2. Proposed System
3. Implementation
4. Experimental Result
5. Related Work and Conclusion

Background

- Illegal accesses are problems
  - Attacks that exploit vulnerabilities of applications

It's difficult to find all vulnerabilities

Sandbox can minimize the damages caused by attacks



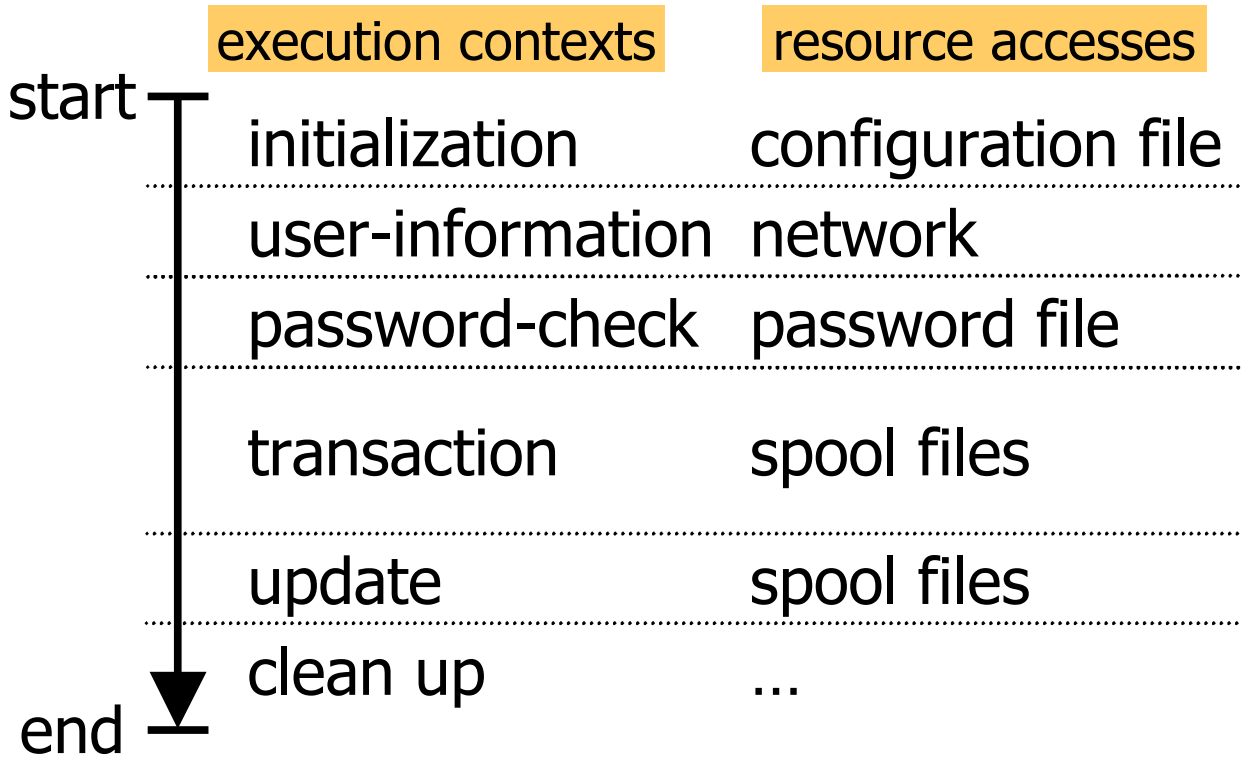
- Monitor the behavior of applications
- Prevent resource accesses that are against intention of users

A policy is a set of specifications of the privileges of programs for operating each resource

Execution Contexts and Resource Accesses

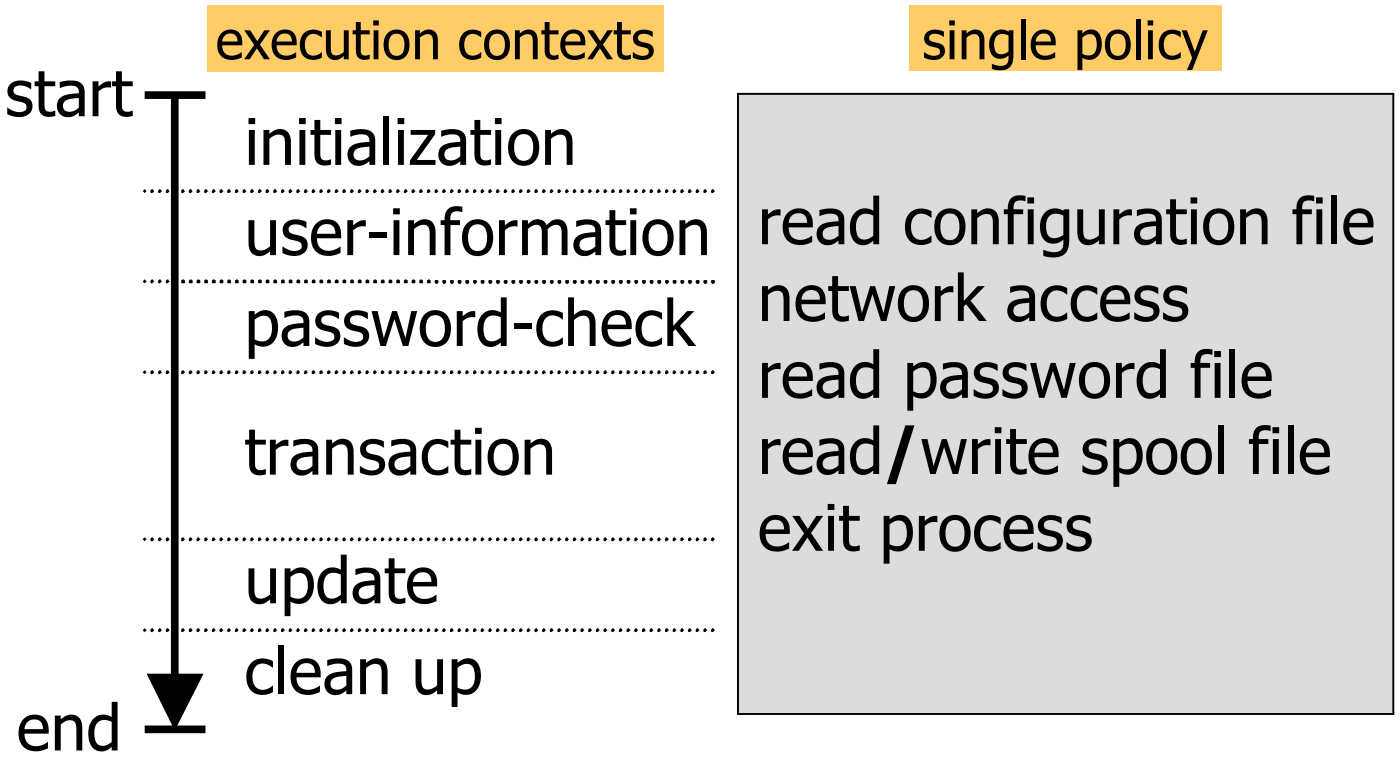
- Execution contexts change with program execution
- Resource accesses change with execution contexts

Ex. POP Server



Problem of Existing Sandboxes

- Only a single policy is applied
  - Against the principle of the least privilege (Provide excessive access rights)



Password file can be accessed in any contexts

Outline

1. Background and Motivation
2. **Proposed System**
3. Implementation
4. Experimental Result
5. Related Work and Conclusion

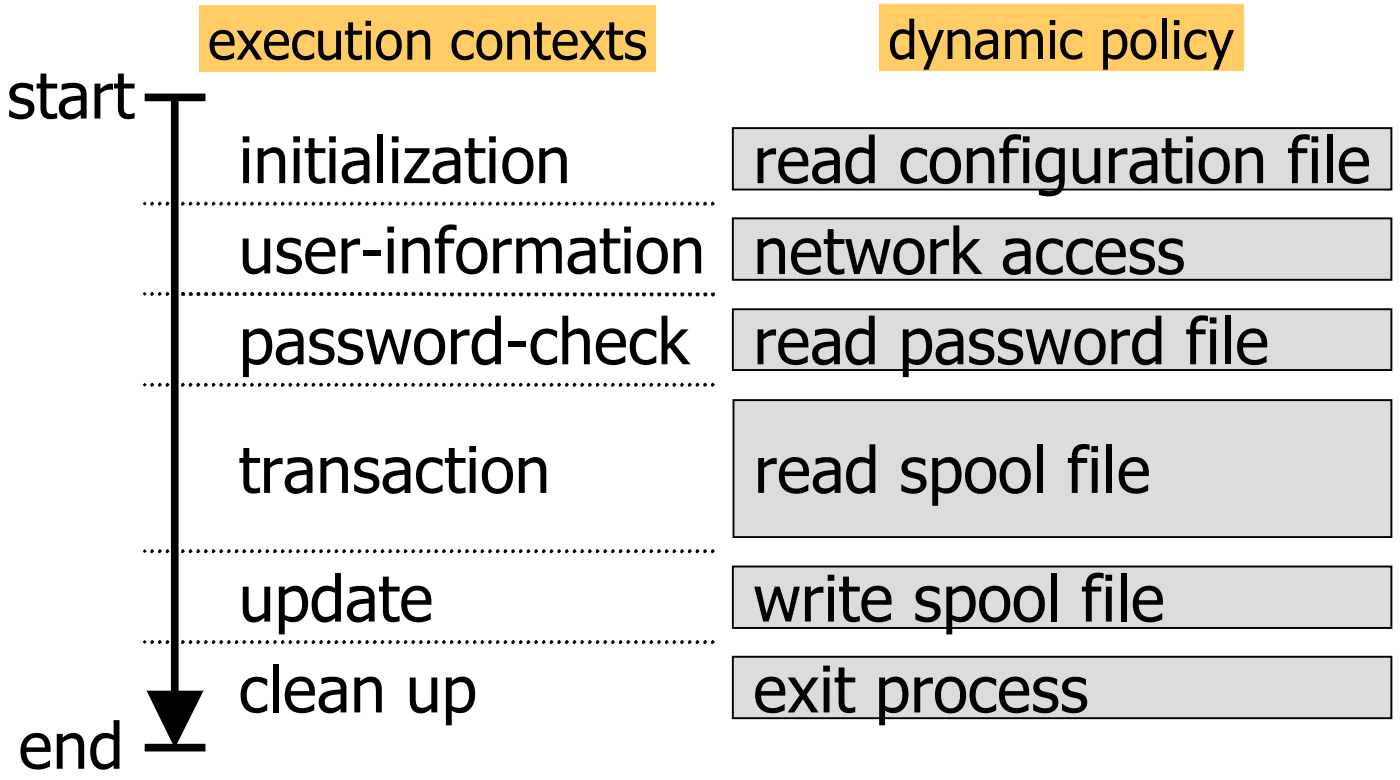
Proposed System

A sandbox system that

- Enables users to dynamically switch between different policies
  - Adequate policy is applied to each execution context

Conforms well to the principle of the least privilege

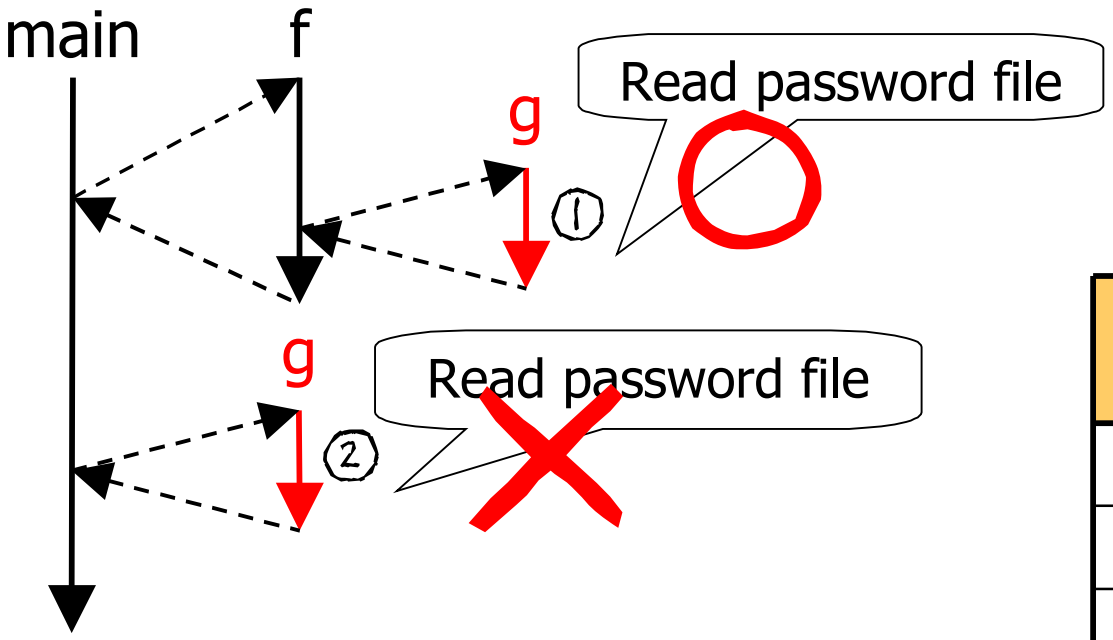
Dynamic Policy Switching



Password file can be accessed only in the password-check context

Approximation of Execution Contexts

- Execution context
  - ≒ A chain of user-defined function calls
    - Each function usually implements some related parts within the application



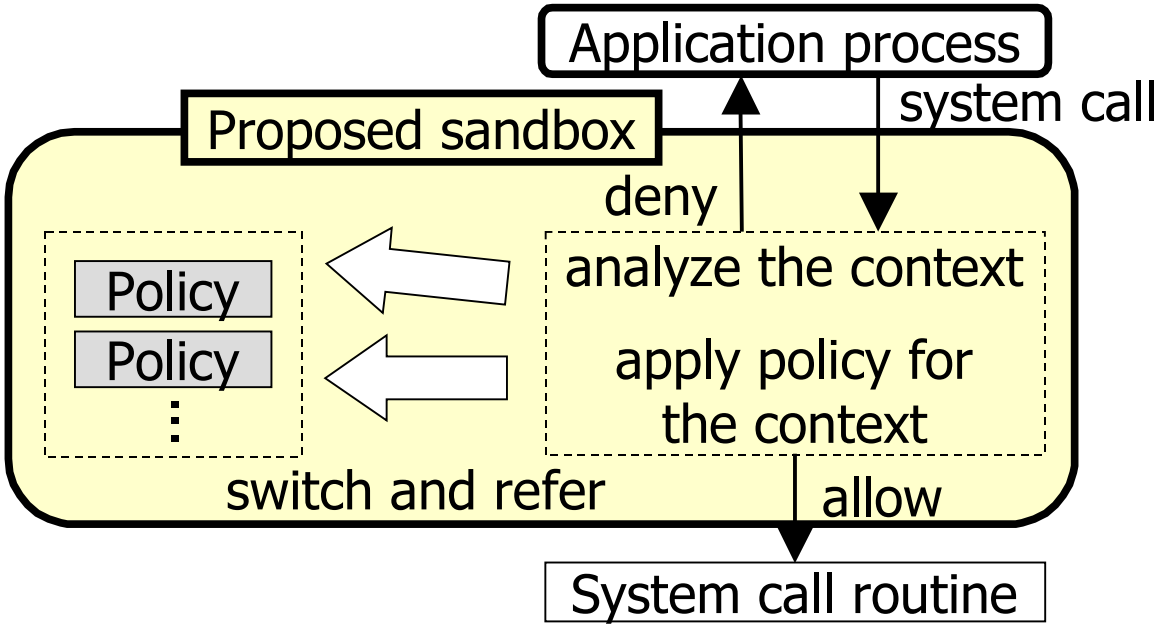
A chain of function call 'g'

①	g - f - main
②	g - main

Function	access to the password file
main	—
f	allowed
g	—

Basic Design

- Policy = Permit/Deny of system calls
- The sandbox
  - Intercepts each system call
  - Analyzes the current execution contexts
  - Determines whether it is allowed or not



Description of Dynamic Policy

Ex. Qpopper 4.0.4

```
#include <sys/fcntl.h>
#include <sys/socket.h>
…
#include "popper.h"
%%
main() {
    socket(AF_INET, SOCK_STREAM, 0);
    fopen("/dev/null", "w+");
    …
}
pop_pass(POP *p) {
    >sleep(_);
    >open(concat("/var/spool/", p->name),
              O_RDWR|O_CREAT, 0666);
    …
}
```

Header

Body

- List of allowed system calls / library functions
- Each system call is allowed to be called from defined functions
  - Default: directly
  - ">": directly or indirectly

Resources that can be dynamically decided by using runtime information

User	Spool File
shioya	/var/spool/shioya
iwasaki	/var/spool/iwasaki

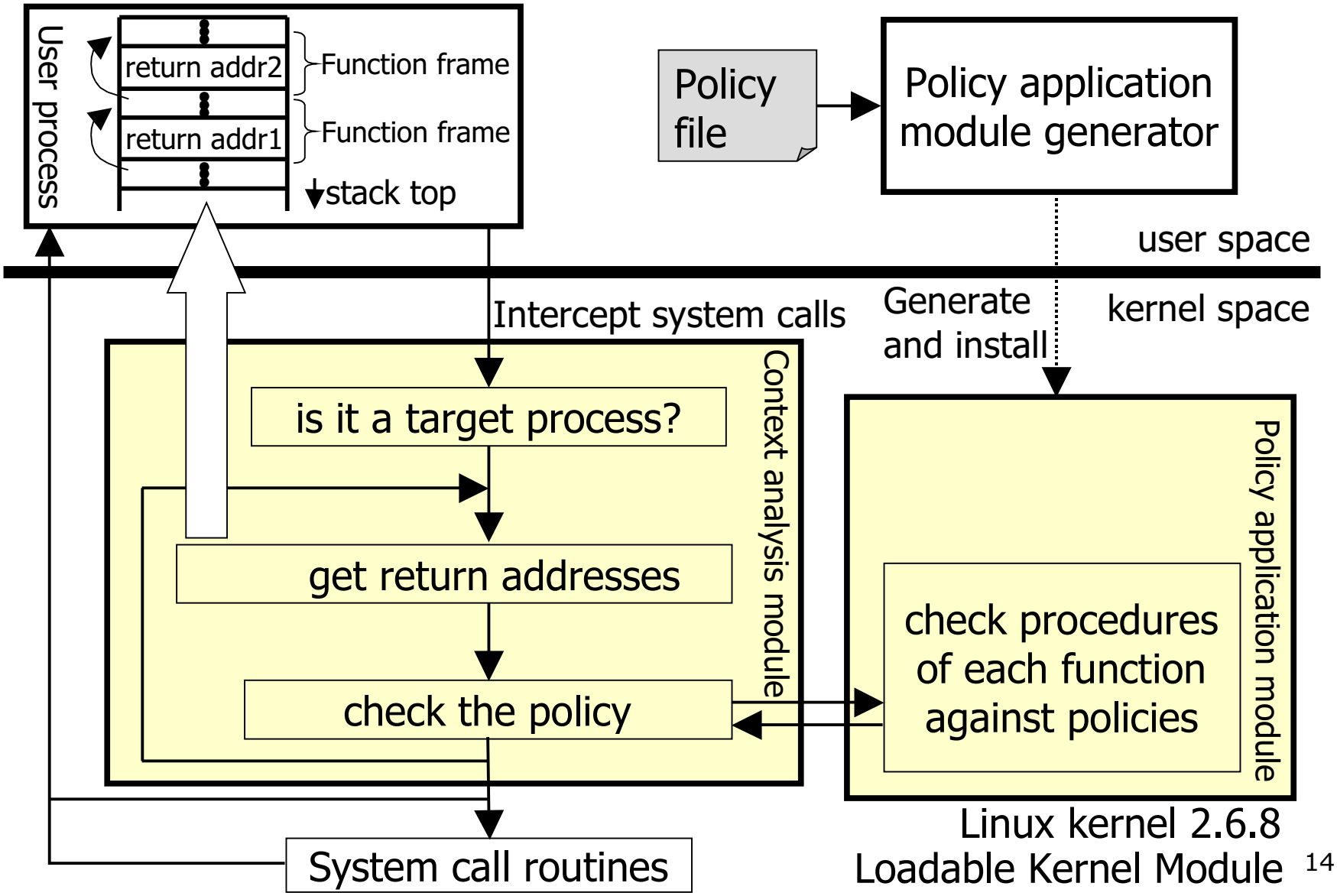
Outline

1. Background and Motivation
2. Proposed System
3. **Implementation**
4. Experimental Result
5. Related Work and Conclusion

Implementation (1/2)

- Implemented on Linux kernel 2.6.8
- Consists of two Loadable Kernel Modules
  - Context analysis module
  - Policy application module
- Generates policy application module from the policy description
- Rewrites a system call entry table to intercept system calls

Implementation (2/2)



Outline

1. Background and Motivation
2. Proposed System
3. Implementation
4. **Experimental Result**
5. Related Work and Conclusion

Experiment

- Detection of attacks
- Overhead of proposed system
  - A micro benchmark
  - Client-side response time

	Server	Client
OS	Linux kernel 2.6.8	Linux kernel 2.4.27
CPU	Pentium 4 3.0-GHz	Pentium III 930-MHz
Mem	1-GB	256-MB

1000BASE-T Ethernet

Detection of Attacks: Qpopper 4.0.4

- Intentional vulnerability for verification
  - Open /etc/passwd if a negative argument is given to a LIST command

```
pop_user() {
    open("/etc/passwd", 0)
    close(_)
    …
}
```

```
pop_list() {
    write(_, _, _)
}
```

- Result of verification
  - Without proposed system: involuntarily opened
  - With proposed system: system call error

The system was able to apply dynamic policies based on execution contexts

Micro Benchmark

- Execution time of operation that consists of opening a file and immediately closing it
  - n is the length of the chain of user-defined functions
  - Analyze n+1 stack frames
  - The case n=0, an approximation of a single-policy sandbox

without proposed system	with proposed system						
	un-sandboxed	sandboxed					
		n = 0	n = 1	n = 2	n = 3	n = 4	n = 5
1.75	1.88	**3.40**	3.79	4.04	4.26	4.57	**4.91**

(μ sec)

The case n=5 extra overhead compared with the case n=0 was 44%

Cleint-side response time

- Measured response time of Qpopper 4.0.4

command	without proposed system	with proposed system	
		un-sandboxed	sandboxed
USER	24.0	24.3	25.0
LIST	11.0	11.2	11.3
RETR	21.8	21.9	22.6

(μ sec)

- Overhead is (compared with without proposed system)
  - un-sandboxed: within 2%
  - sandboxed: almost 4%

The overhead is not a serious problem compared with the network latency

Outline

1. Background and Motivation
2. Proposed System
3. Implementation
4. Experimental Result
5. **Related Work and Conclusion**

Related Work

- Source codes of target applications must be modified
  - SubDomain [Cowan et al. 00]
    - Can switch policies when a target process calls an exec system call or a special system call change_hat
  - An extension of Systrace [Kurchuk et al. 04]
    - Can switch policies when a special function

Our system does not require the application code modification

- The target is not native codes
  - Java Stack Inspection [Wallach et al. 97]
    - The mechanism to switch policies in Java sandboxes

Our system achieves dynamic policy switches for native code

Conclusion

- We proposed a sandbox system that can apply dynamic policies in accordance with the execution contexts
- It uses a chain of user-defined function calls as an approximation of an execution context
- We implemented on Linux and evaluated effectiveness by experiments