Process Hijacking

Victor Zandy, Barton P. Miller, Miron Livny
University of Wisconsin - Madison

March 25, 1999

What is Hijacking?

Your Process

What is Hijacking?

Your Process
with
Checkpointing

Process Hijacking turns any running process into a Condor job

- You can migrate your process to another host while it runs

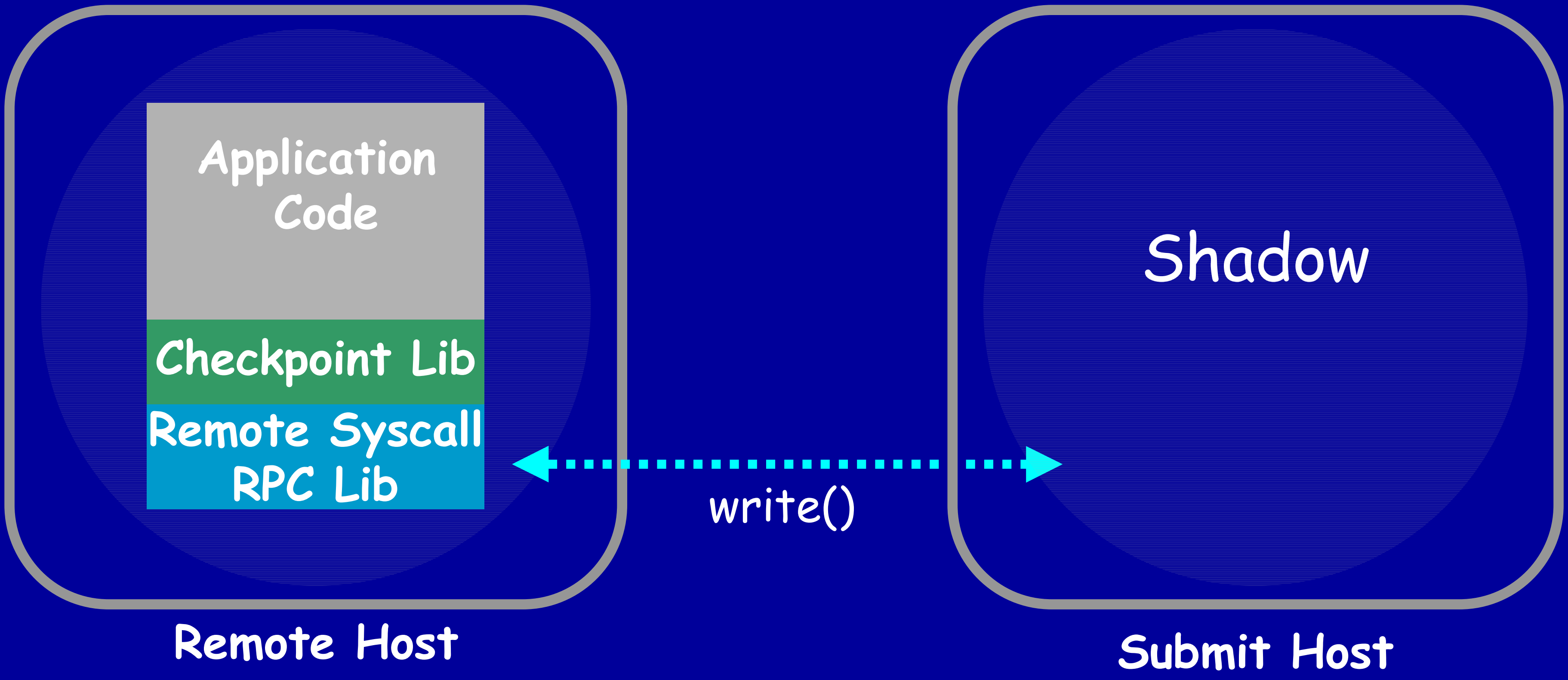
What is Condor?

Condor is a system for high-throughput distributed computing on a heterogeneous network

A Condor job can be migrated

- checkpointing saves the process state
- system call RPCs allow remote I/O

A Condor Job in Execution



The Hijacking Problem

Preparing a Condor job is convenient
- No re-programming
- Just re-link the executable to install the remote system call and checkpointing library
- Re-linking is automatic with condor_compile

However, re-linking is not always possible
- No access to object (.o) files
- The program is already running!

Process Hijacking

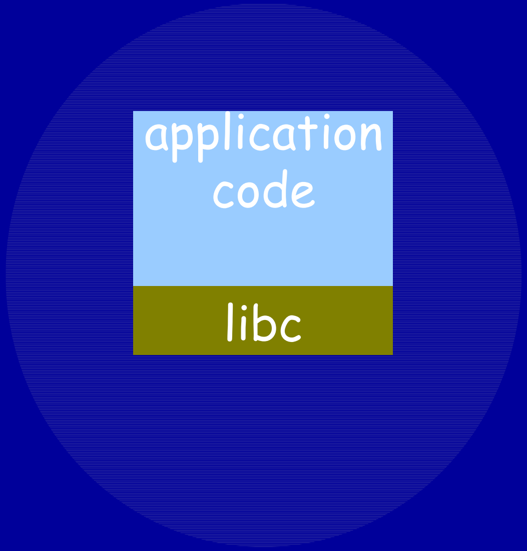
Process hijacking eliminates the need to re-link and it turns any running process into a Condor job

It is as convenient as Condor:
- **`hijack PID`**

The hijacker uses the DynInst API to
- inject the system call and checkpoint libraries
- replace the original system calls with RPC stubs

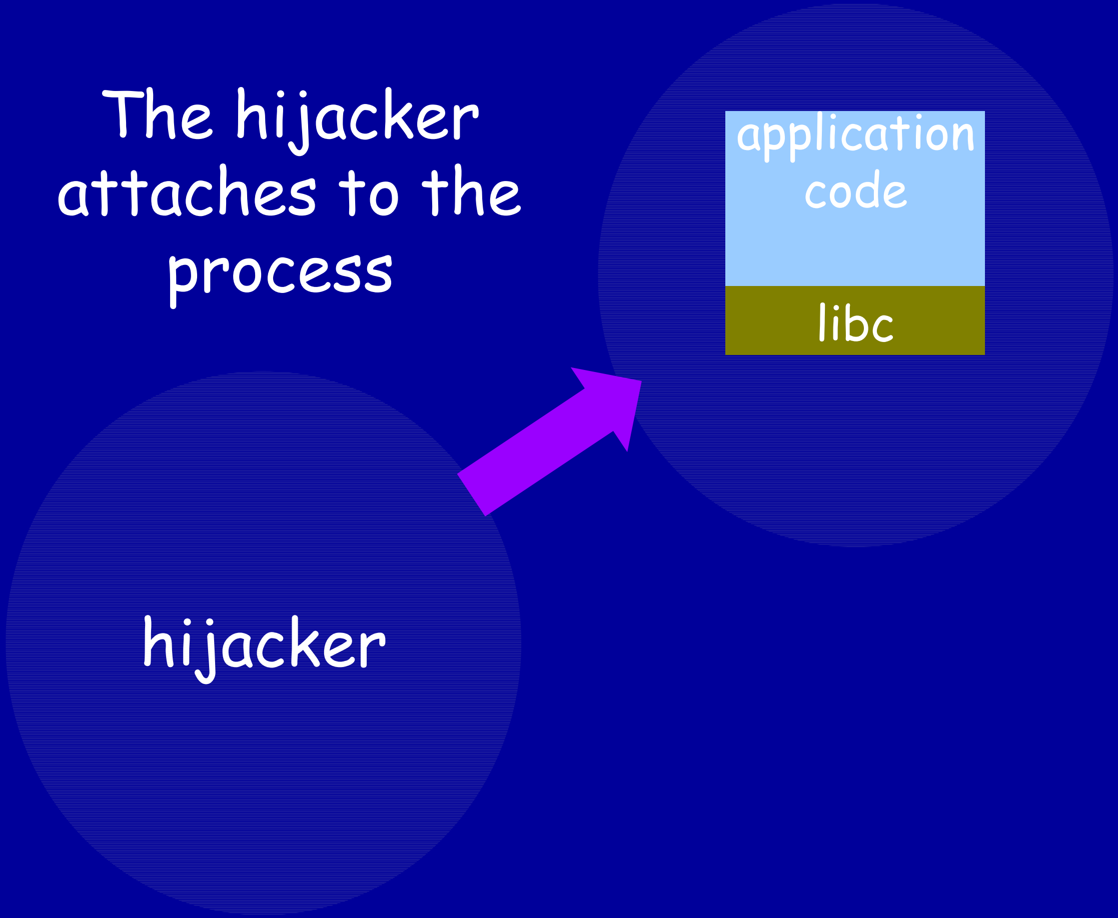
Process Hijacking

The ordinary process
is minding its
own business



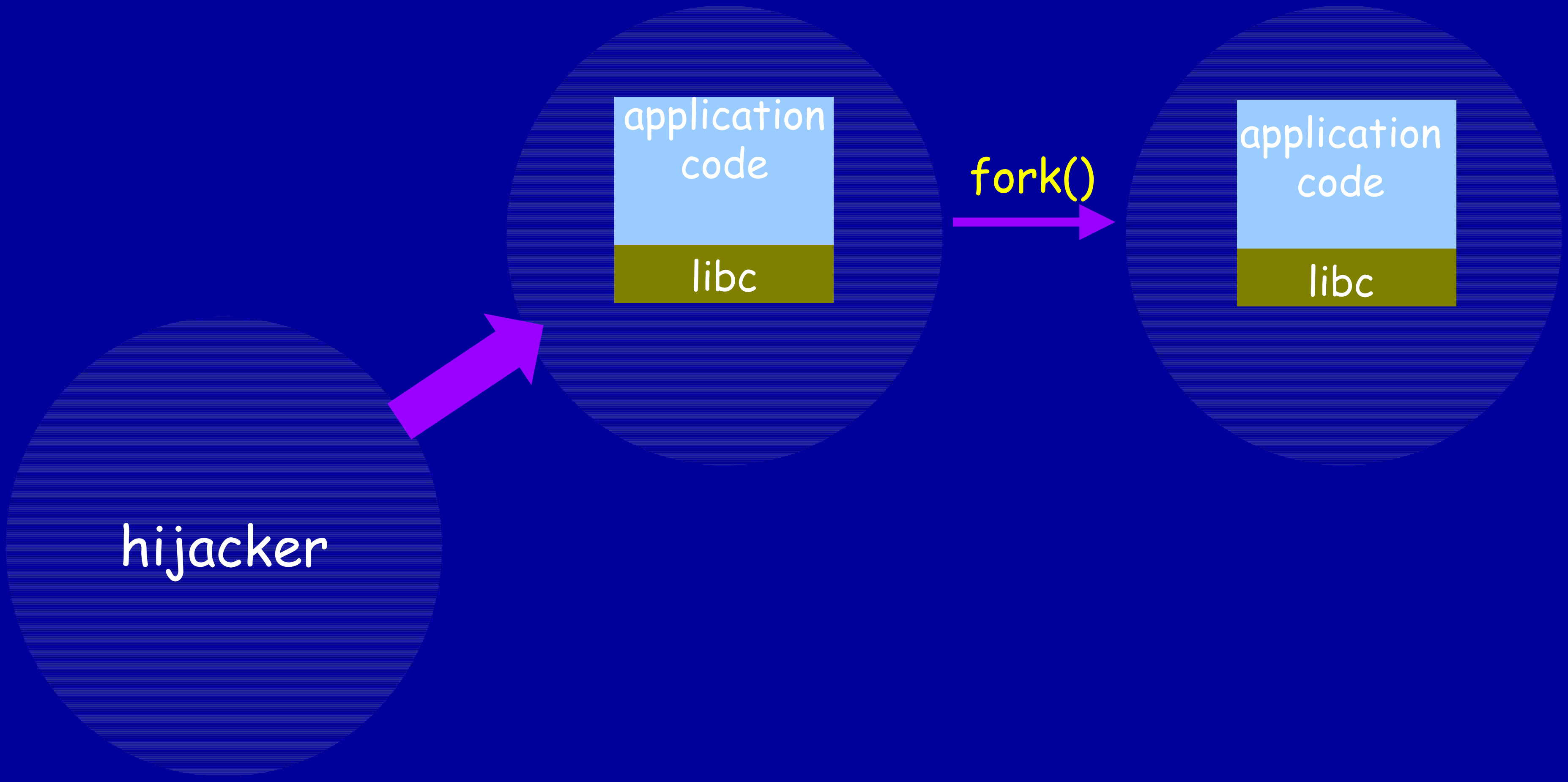
Process Hijacking

The hijacker attaches to the process

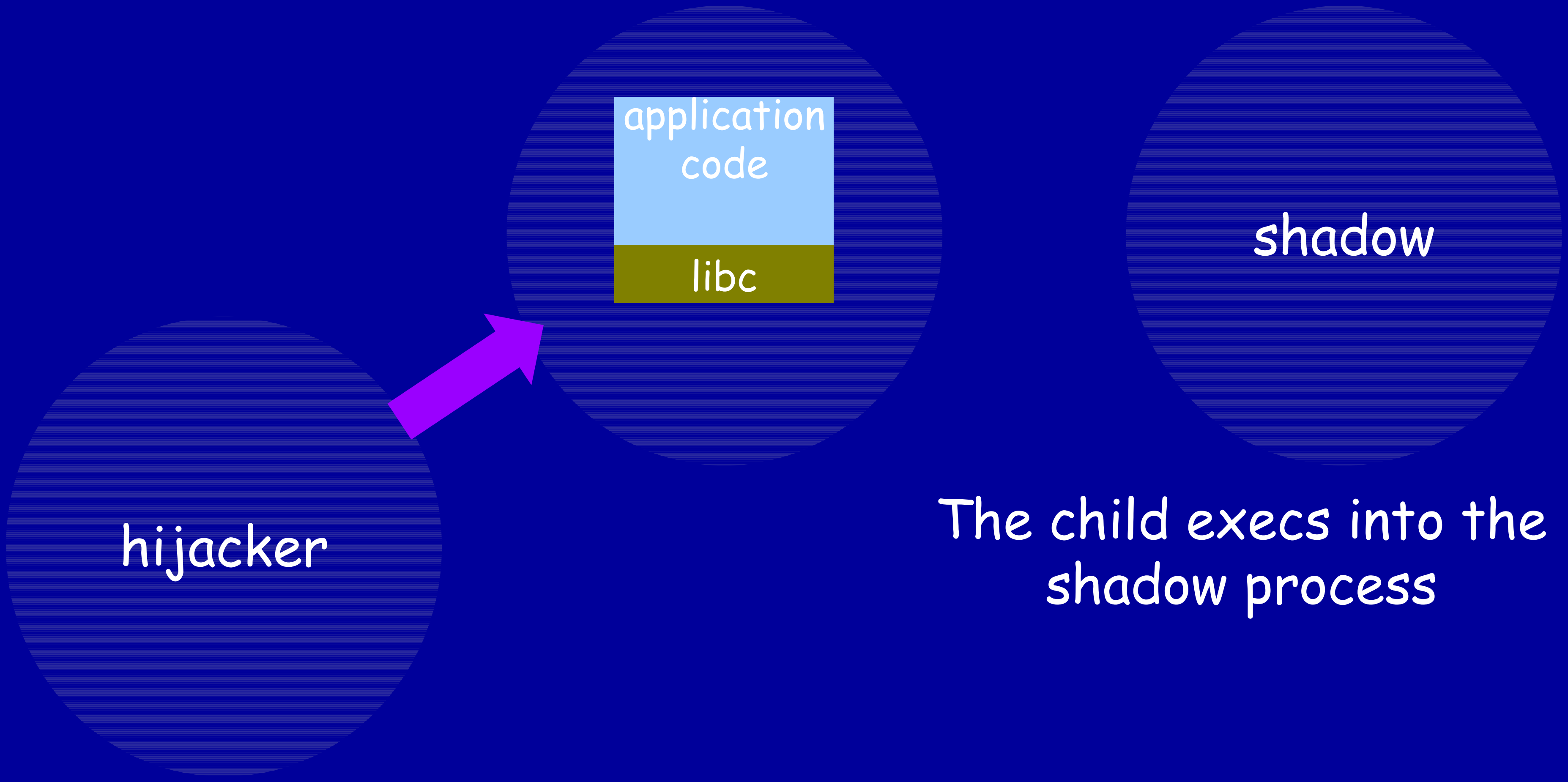


Process Hijacking

The hijacker makes the process fork



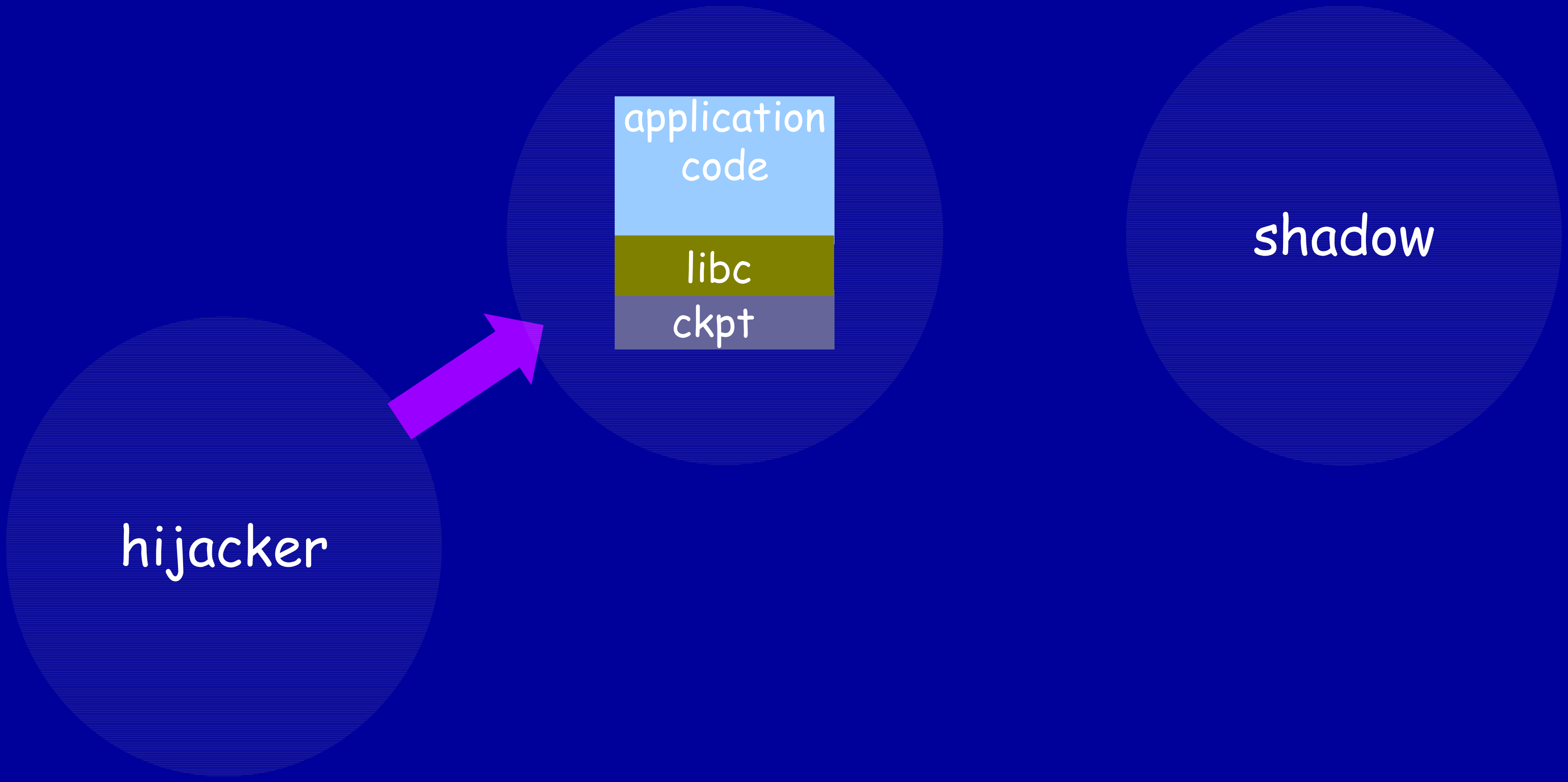
Process Hijacking



The child execs into the shadow process

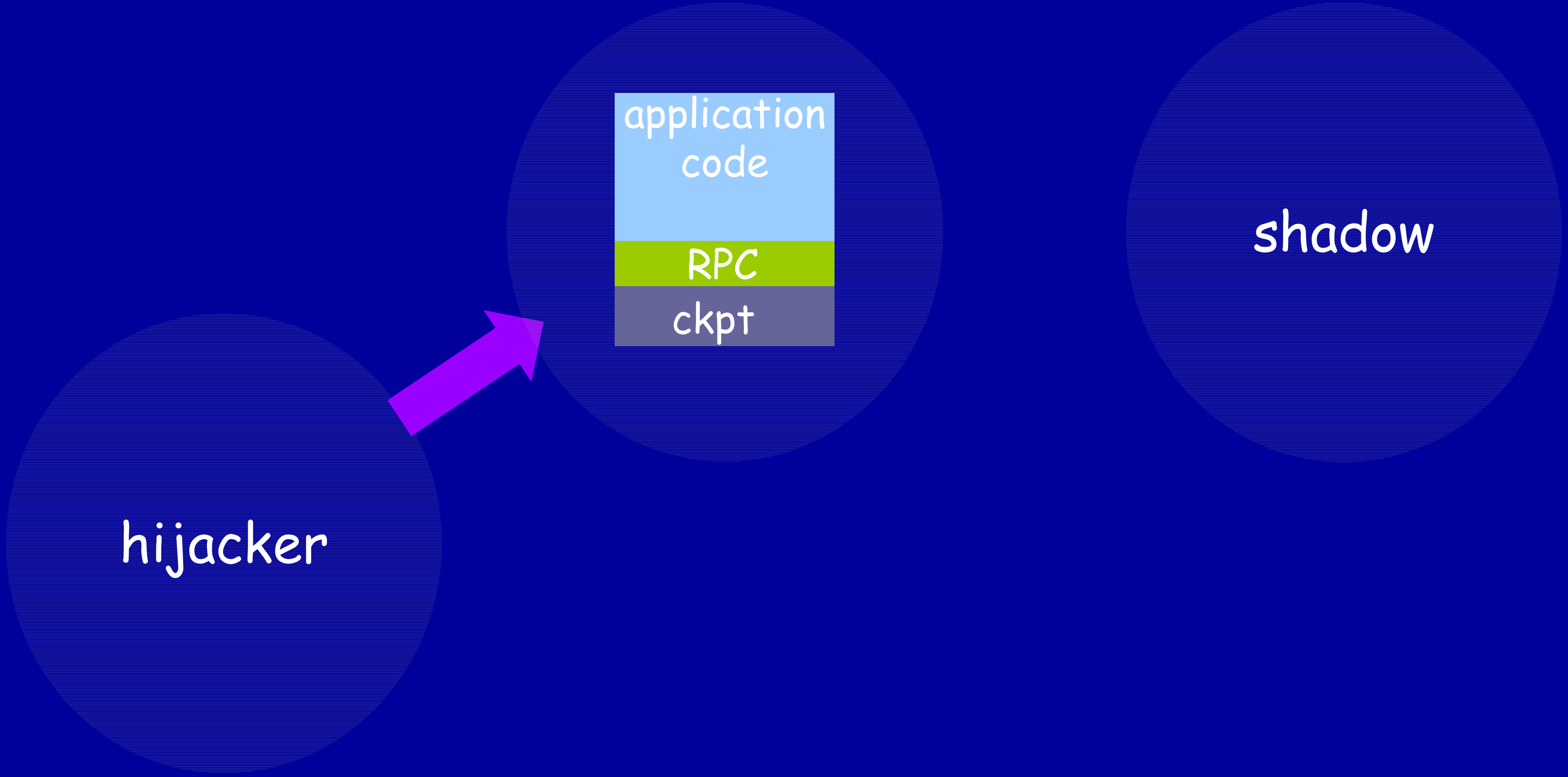
Process Hijacking

The hijacker loads checkpoint and remote system call libraries into the process

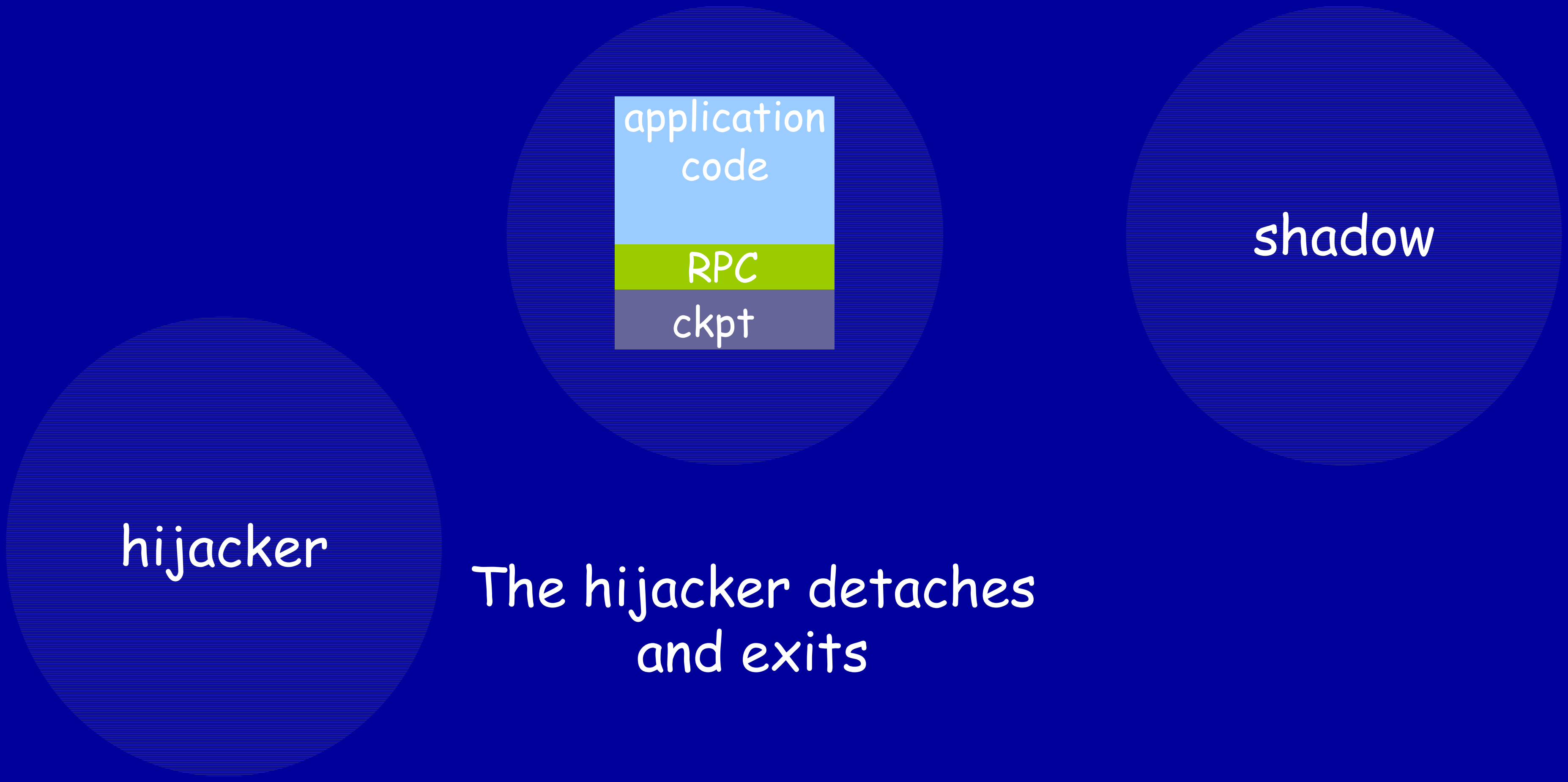


Process Hijacking

The hijacker replaces the libc system calls with calls to the Condor RPCs

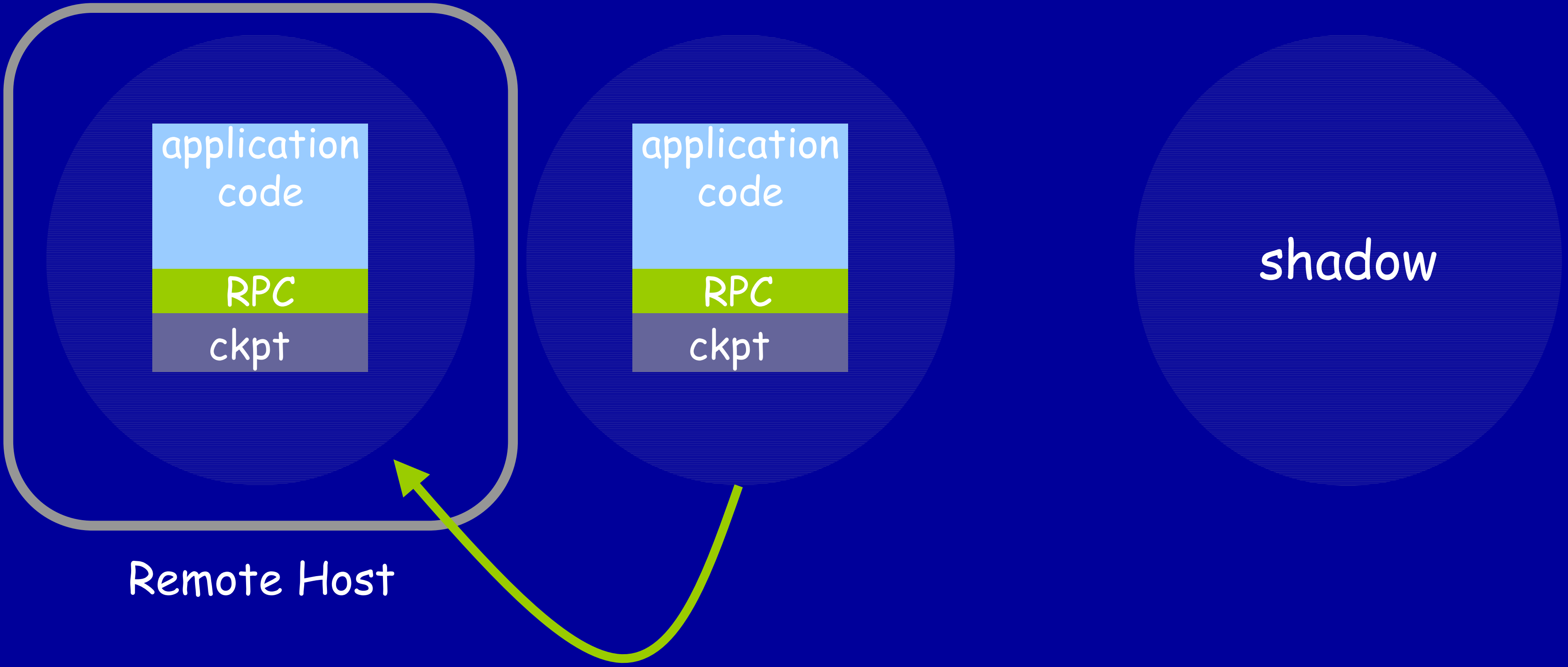


Process Hijacking



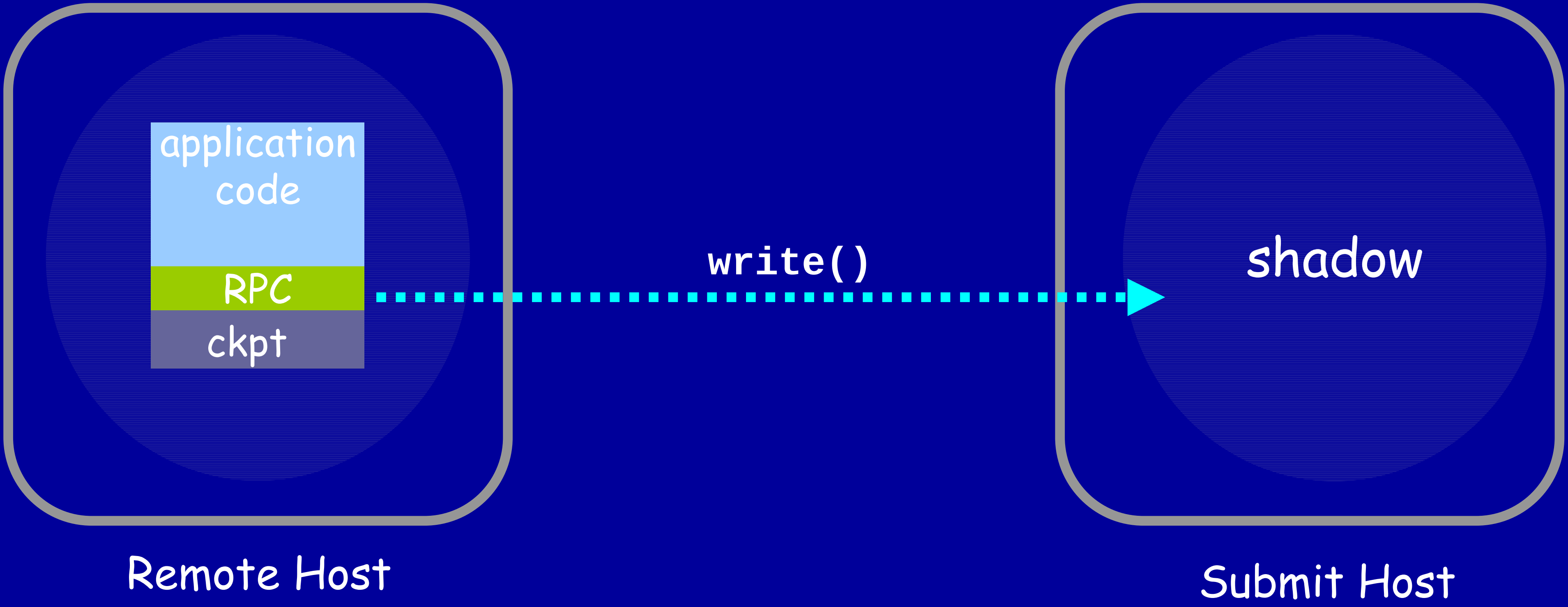
The hijacker detaches and exits

Process Hijacking



The process migrates to a remote host

Process Hijacking

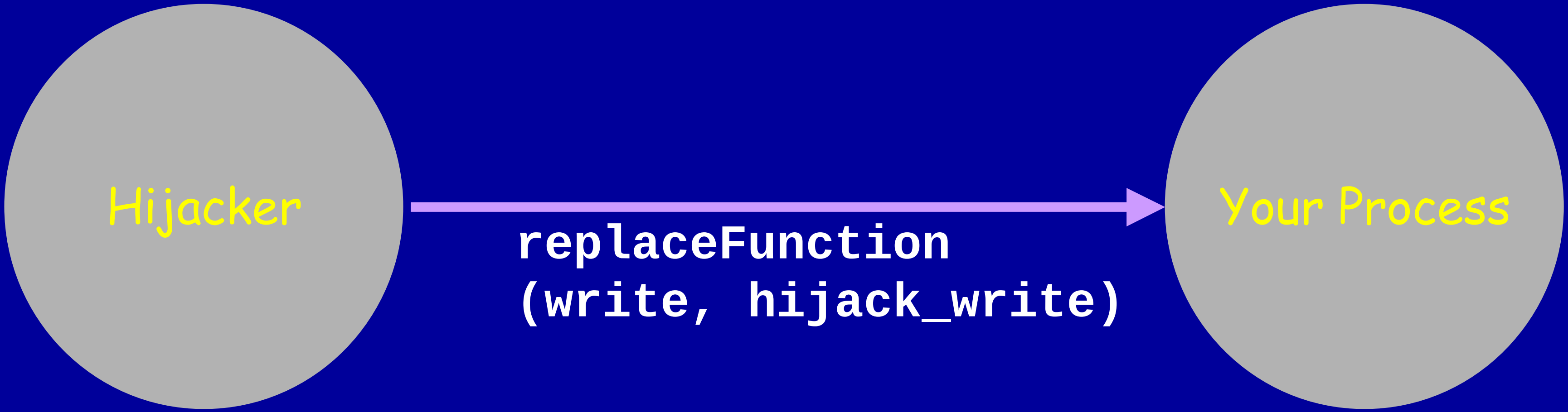


System calls are now RPCs to the shadow

Replacing System Calls

A new DynInst call replaces the system calls of the hijacked process

```
replaceFunction(oldfunc, newfunc)
```

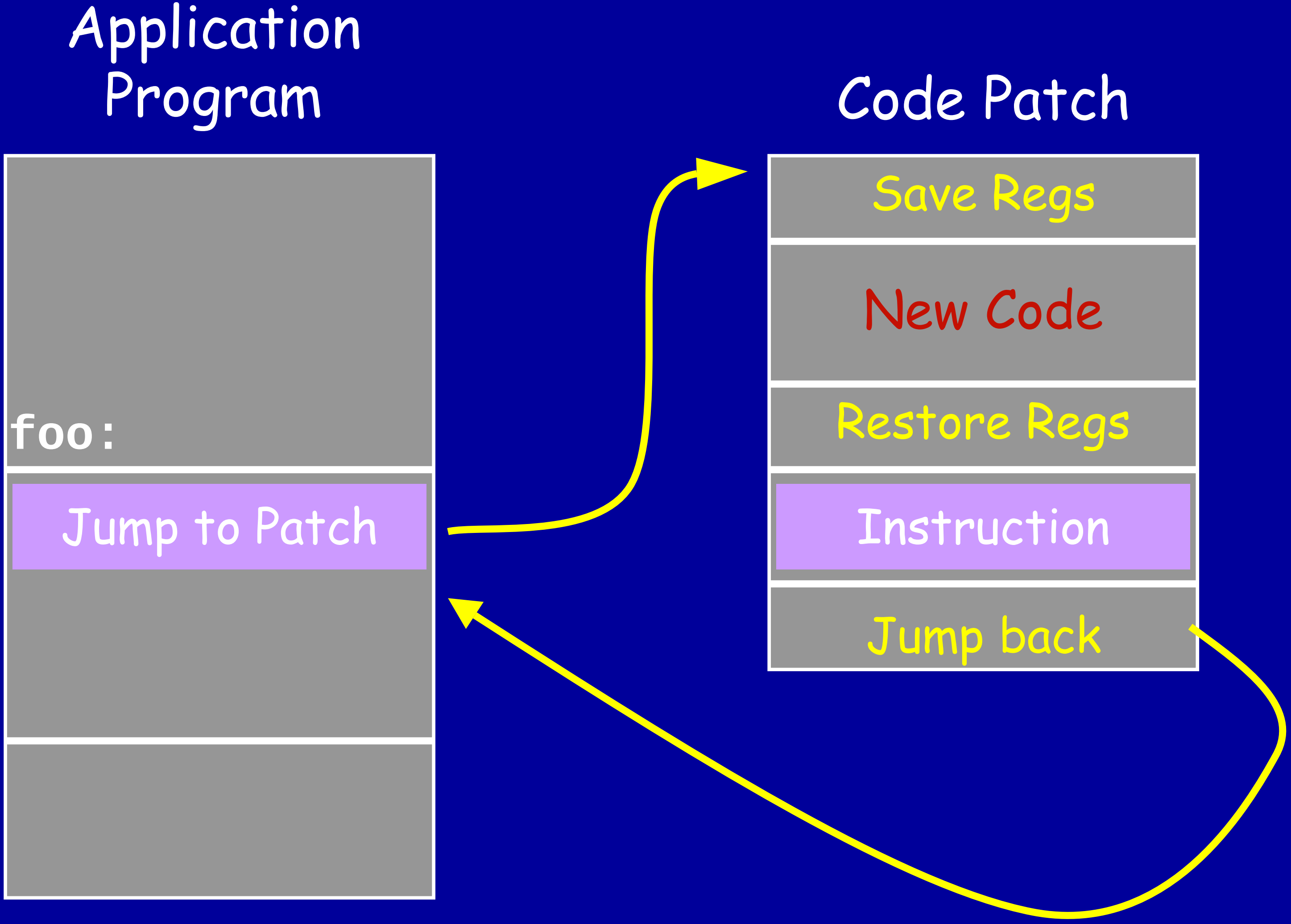


```
replaceFunction
(write, hijack_write)
```

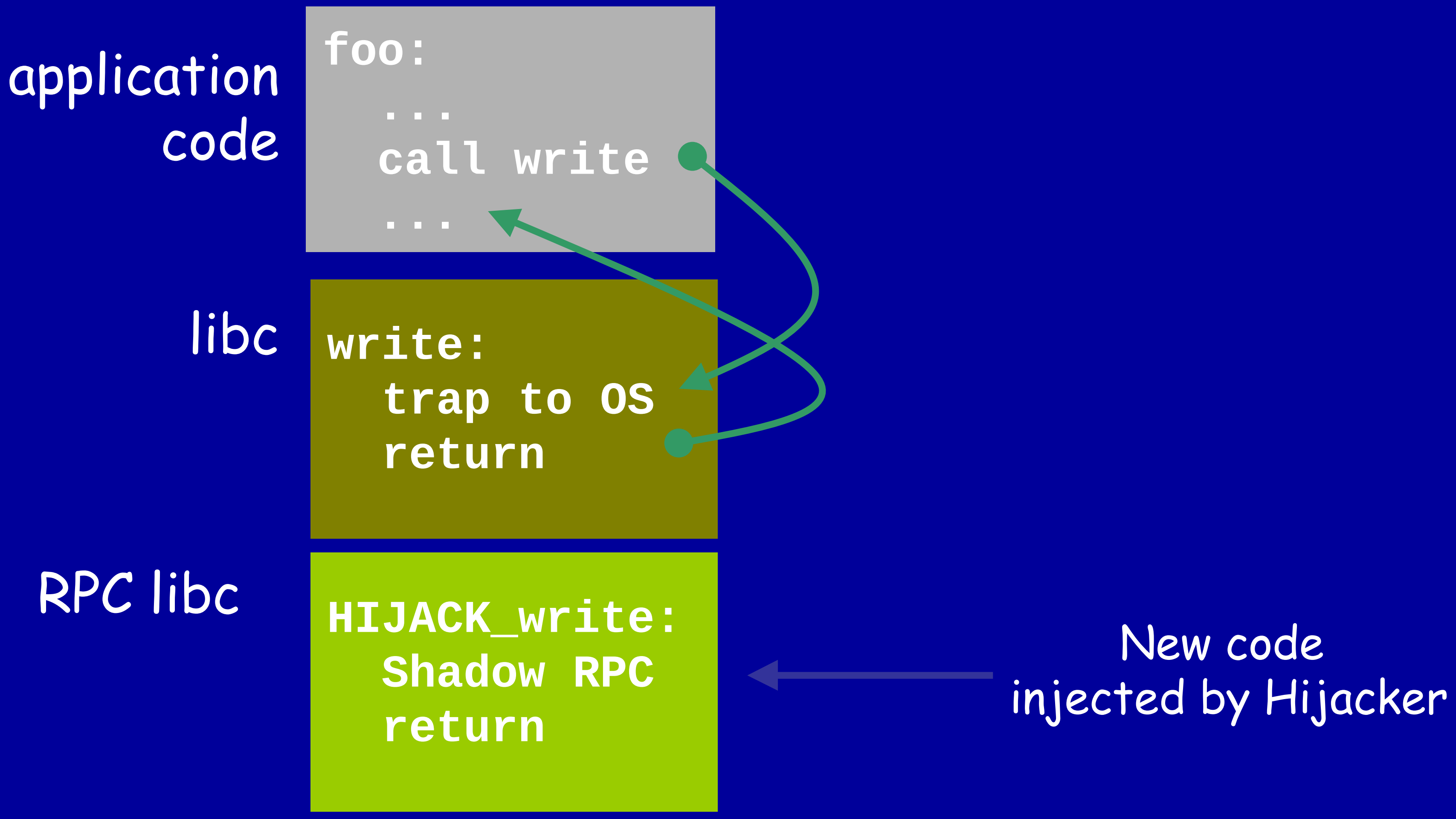
Patching in New Code

Application Program
foo:
Instruction

Patching in New Code



Replacing System Calls



application code

```
foo:
   ...
   call write
   ...
```

libc

```
write:
   trap to OS
   return
```

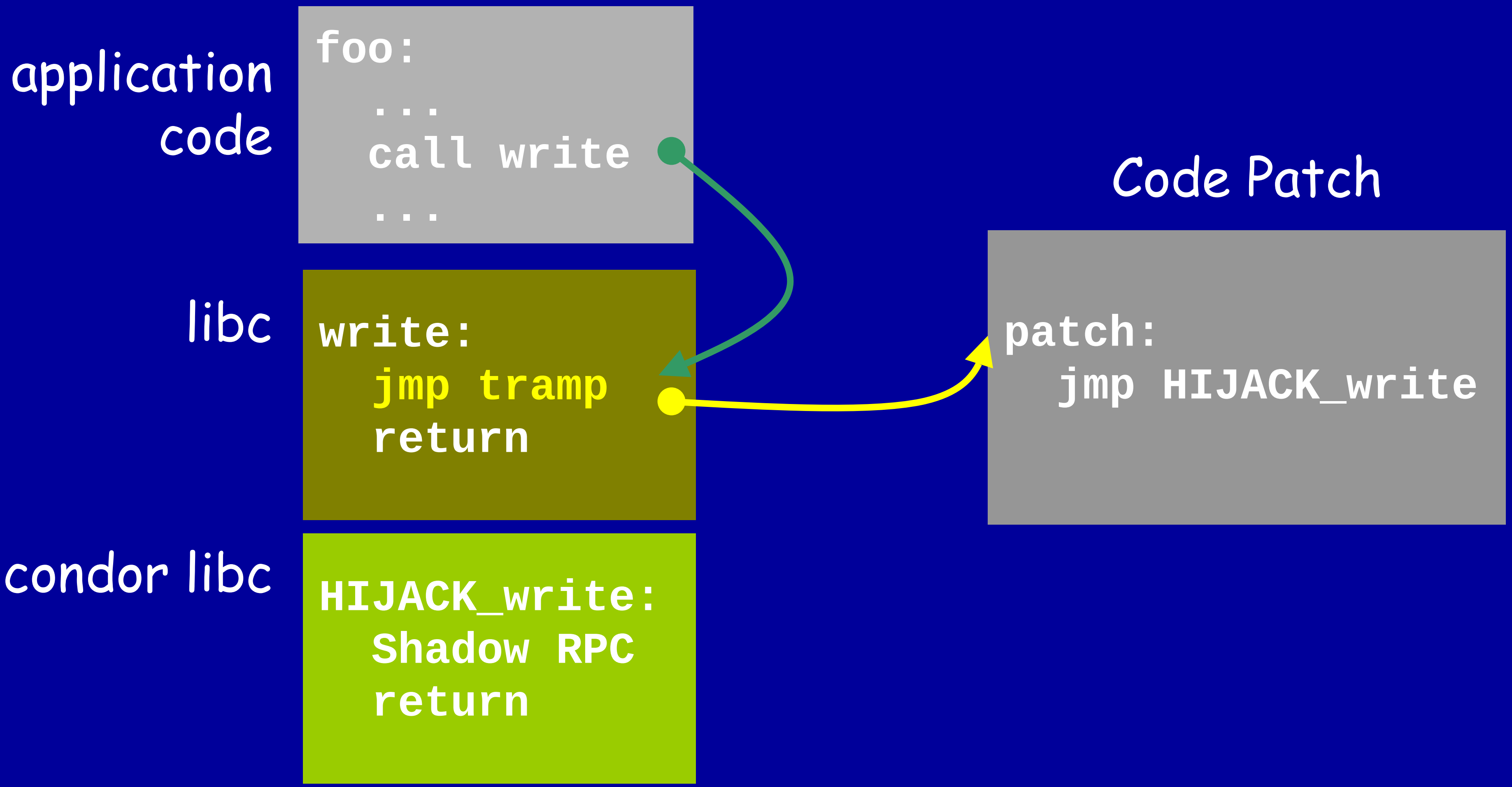
RPC libc

```
HIJACK_write:
   Shadow RPC
   return
```

New code injected by Hijacker

Your process during the hijack

Replacing System Calls



application code

```
foo:
   ...
   call write
   ...
```

libc

```
write:
   jmp tramp
   return
```

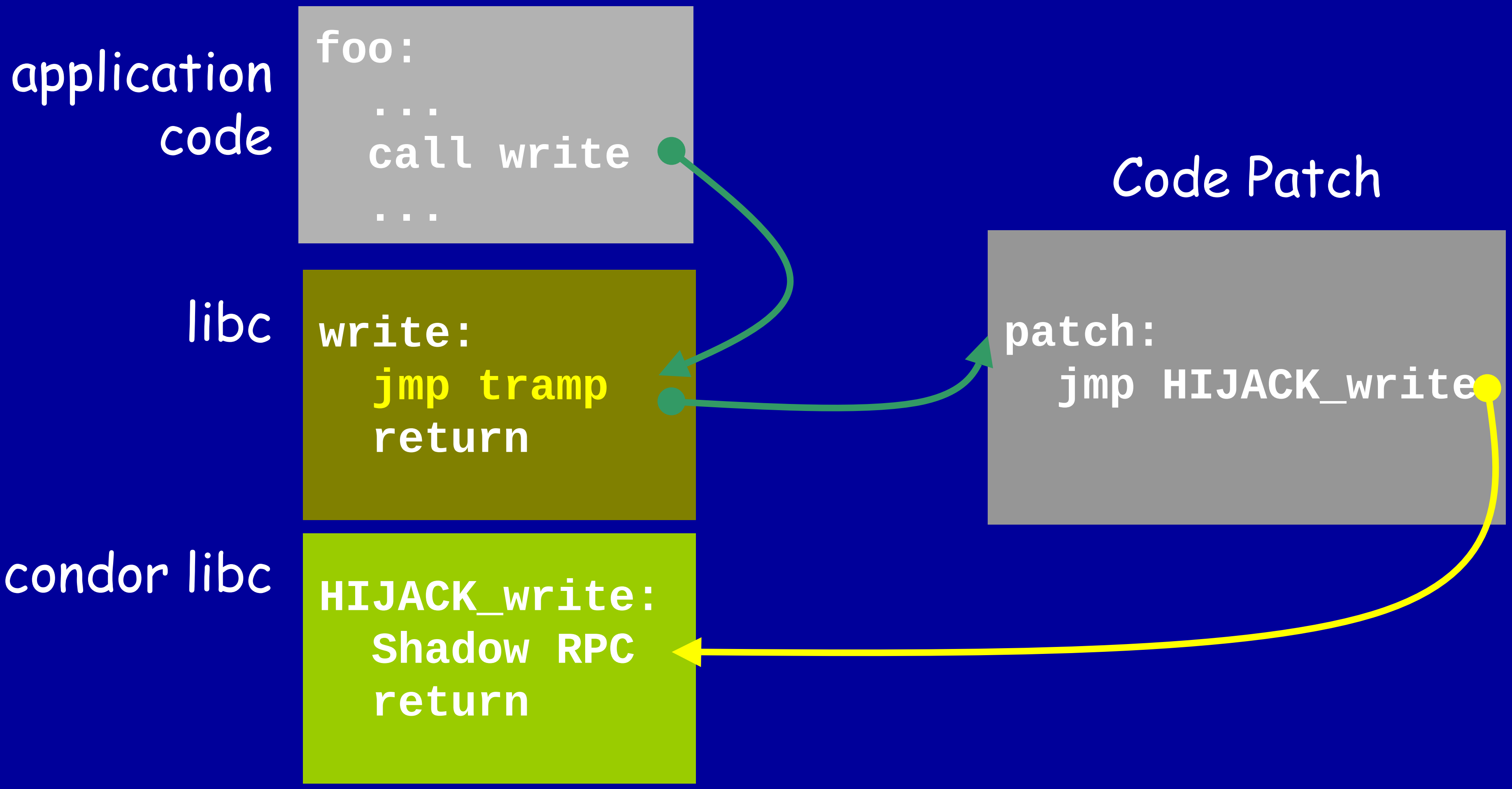
condor libc

```
HIJACK_write:
   Shadow RPC
   return
```

Code Patch

```
patch:
   jmp HIJACK_write
```

Replacing System Calls



application code

```
foo:
   ...
   call write
   ...
```

libc

```
write:
   jmp tramp
   return
```

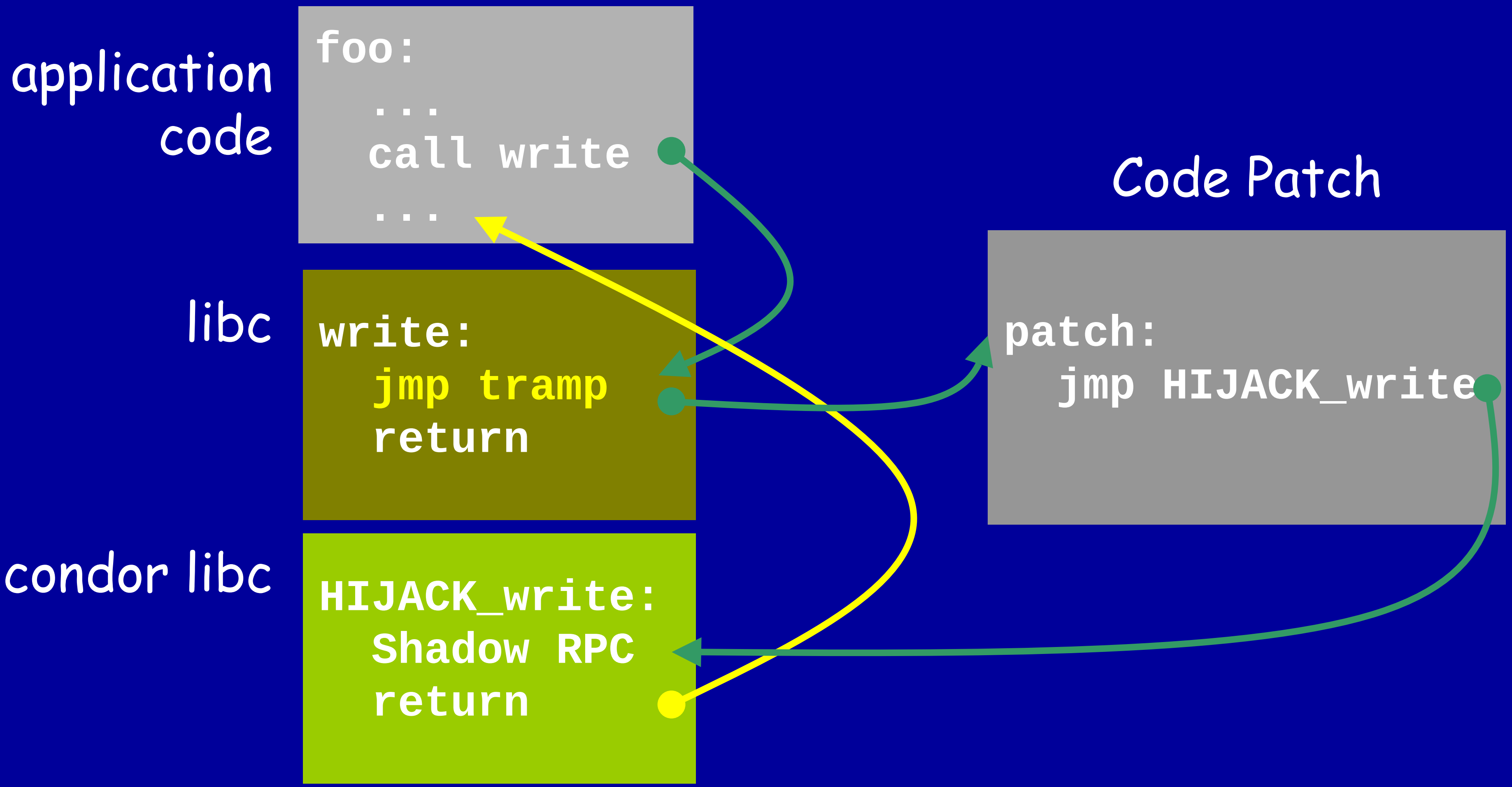
condor libc

```
HIJACK_write:
   Shadow RPC
   return
```

Code Patch

```
patch:
   jmp HIJACK_write
```

Replacing System Calls



application code

```
foo:
   ...
   call write
   ...
```

libc

```
write:
   jmp tramp
   return
```

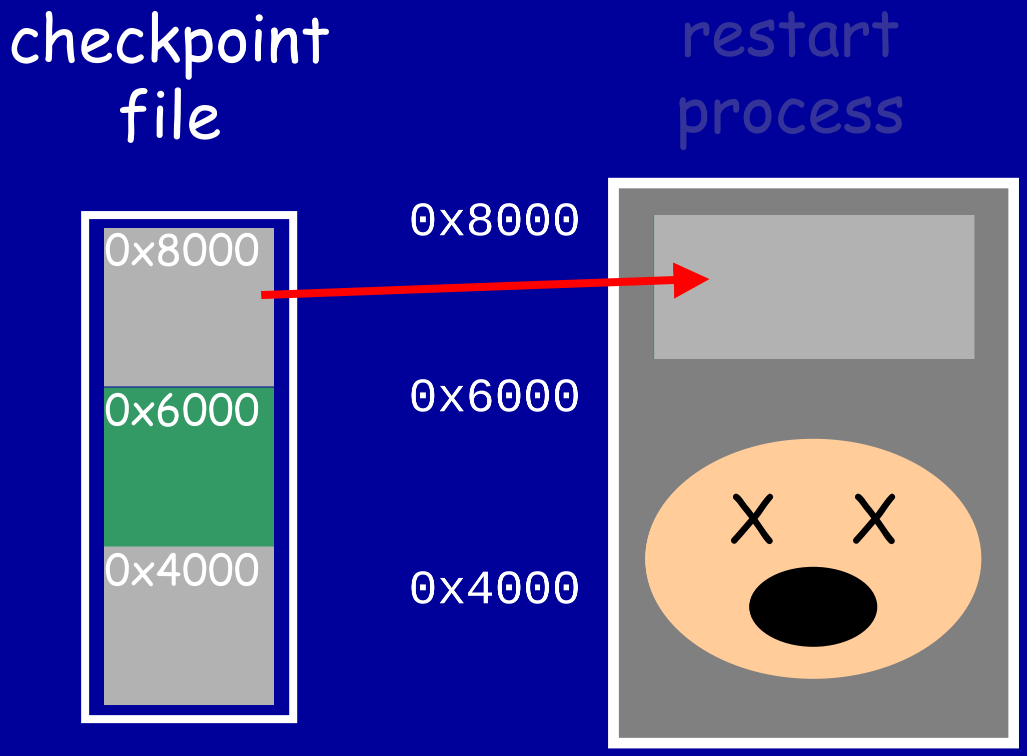
condor libc

```
HIJACK_write:
   Shadow RPC
   return
```

Code Patch

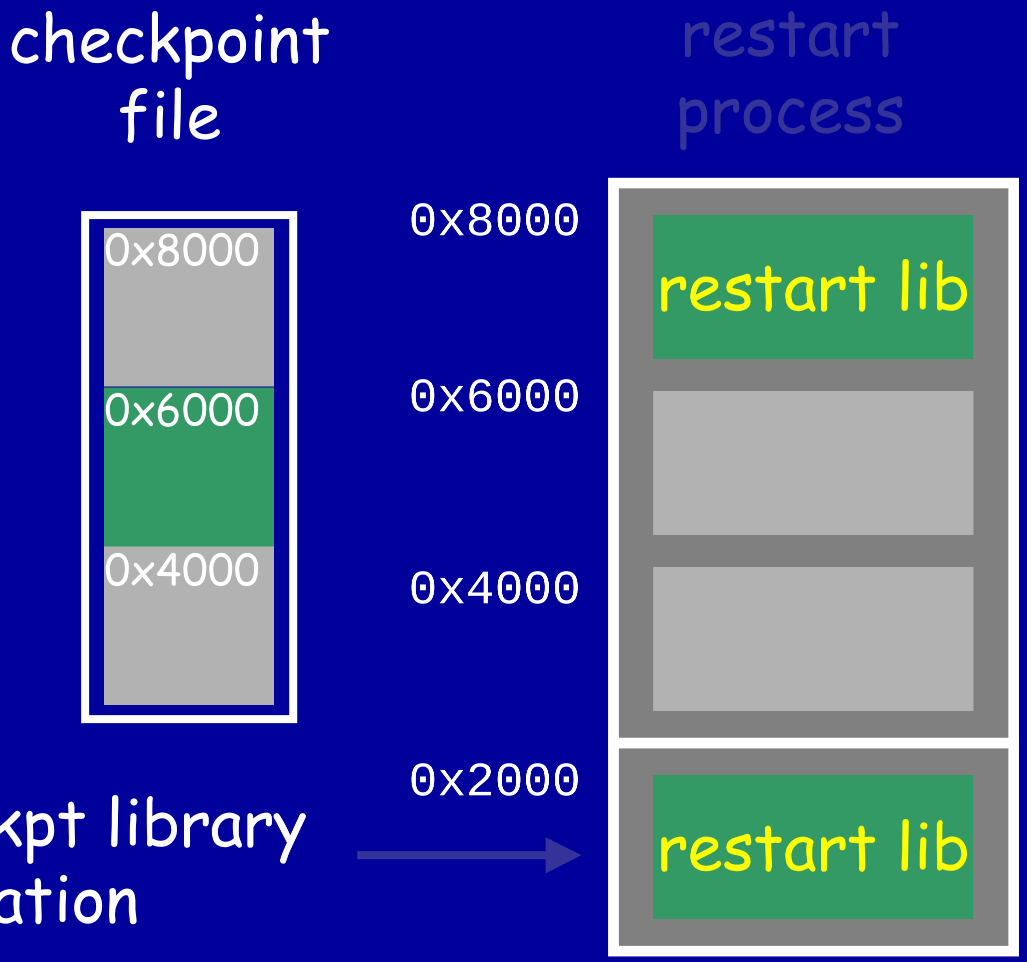
```
patch:
   jmp HIJACK_write
```

Restarting a Checkpoint



Problem: copying from the checkpoint file might clobber the (executing) ckpt library.

Restarting a Checkpoint



Solution: Reload the ckpt library to a safe location

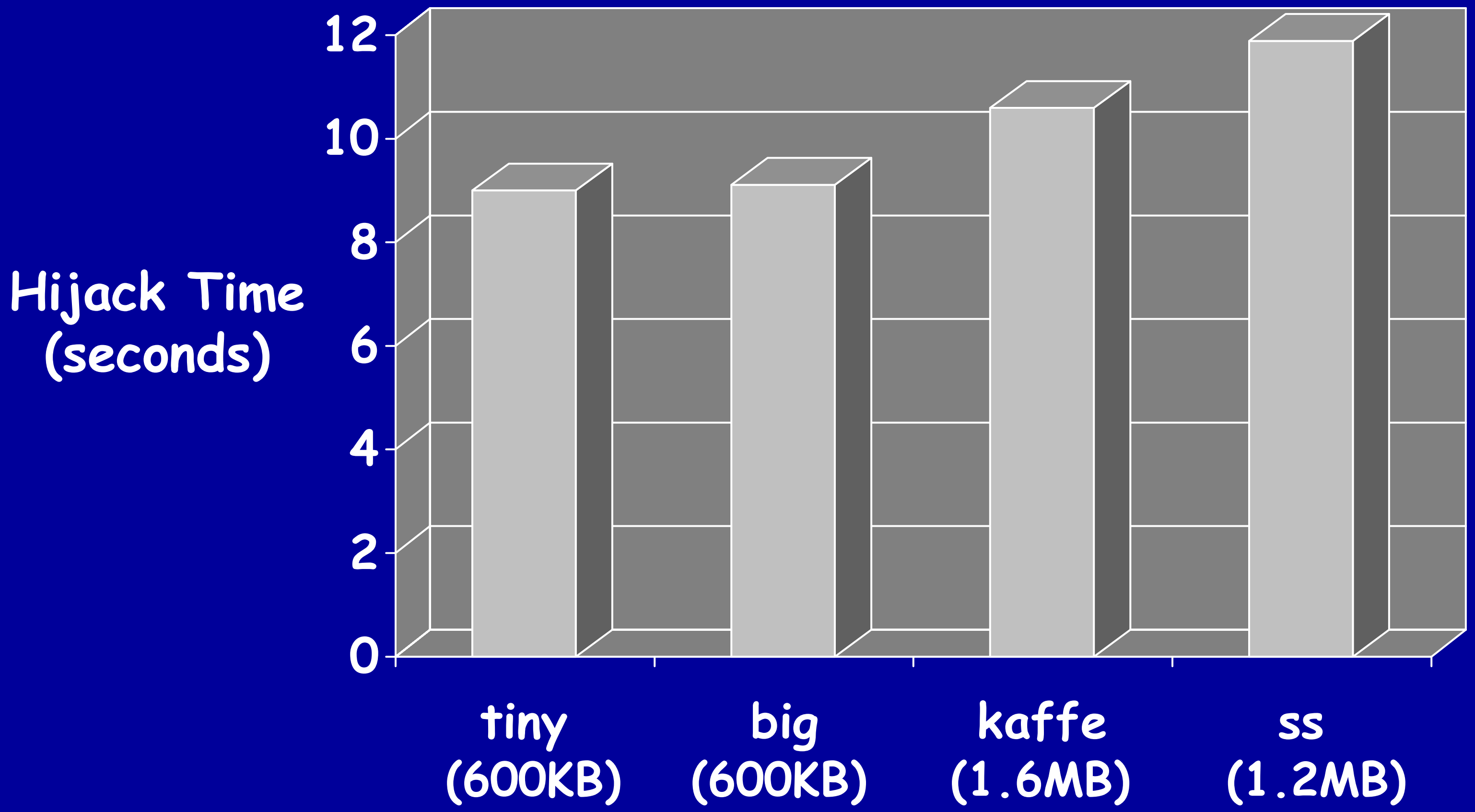
Status

Process hijacking has been implemented for Sparc Solaris 2.5.1

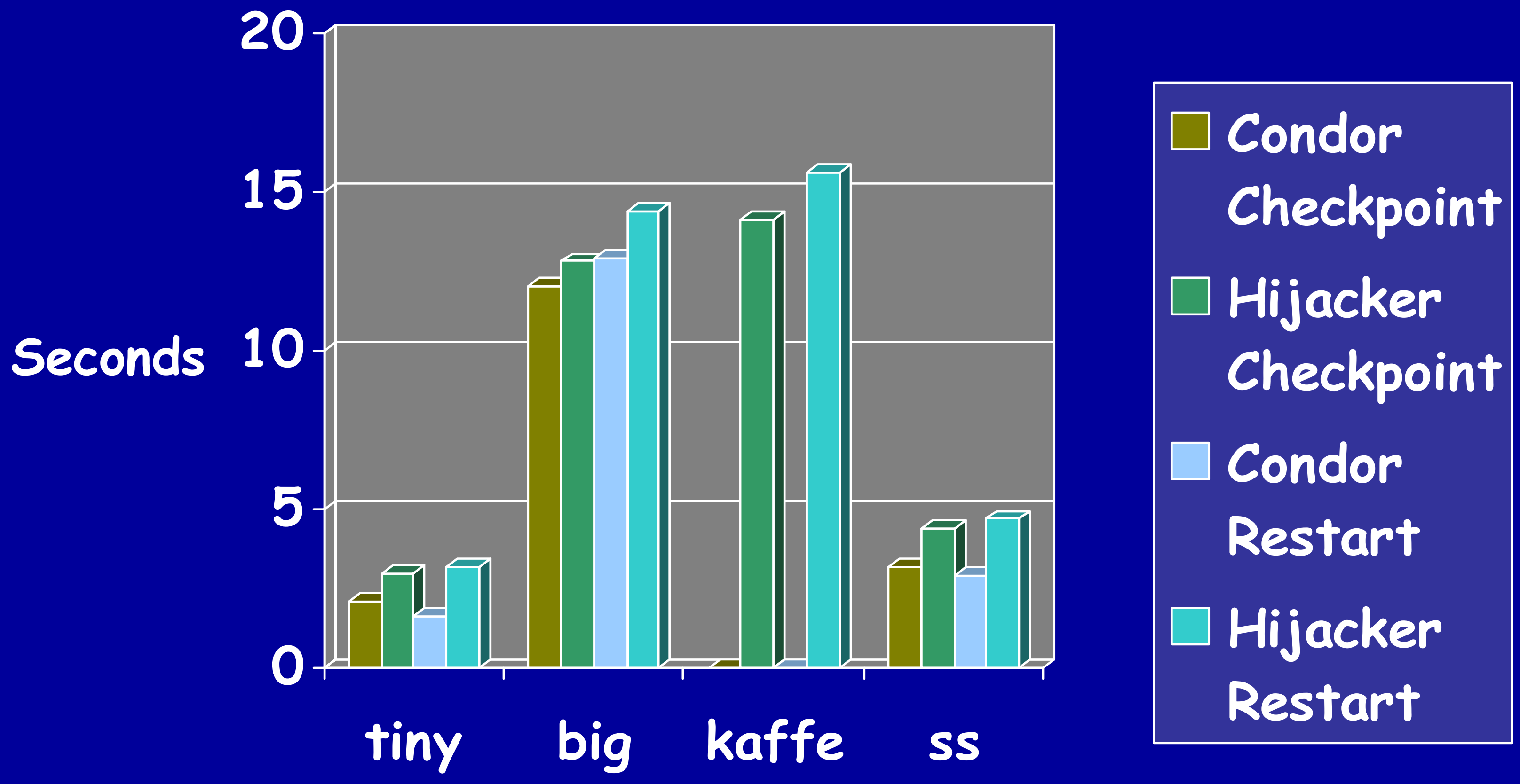
Hijacker	700 lines
Checkpoint	500 lines
Syscalls (Condor)	25K lines

We are able to hijack an unmodified Java VM running a real, compute-intensive Java application (4400 lines)

Hijack Cost


Hijack Time (seconds)
12
10
8
6
4
2
0
tiny (600KB)
big (600KB)
kaffe (1.6MB)
ss (1.2MB)

Migration Costs



Hijacker checkpoints are 1.5M larger than Condor's

Limitations

The process hijacker inherits Condor's checkpointing limitations

- No sockets
- No kernel-level threads
- One process

Limitations

Unlike Condor, our shadow process must not exit until the program is done

- File state acquired before hijack time cannot be checkpointed

If the shadow dies, the job is lost

Summary

Condor provides high-throughput computing that is convenient to both application users and resource owners

Process Hijacking enhances Condor by allowing any process to become a Condor job, without advance preparation

Hijacking is a demonstration of the power and utility of runtime code modification with the DynInst API

Change calls with DynInst

DynInst is an API for changing the code of a running process

We use DynInst in three ways:

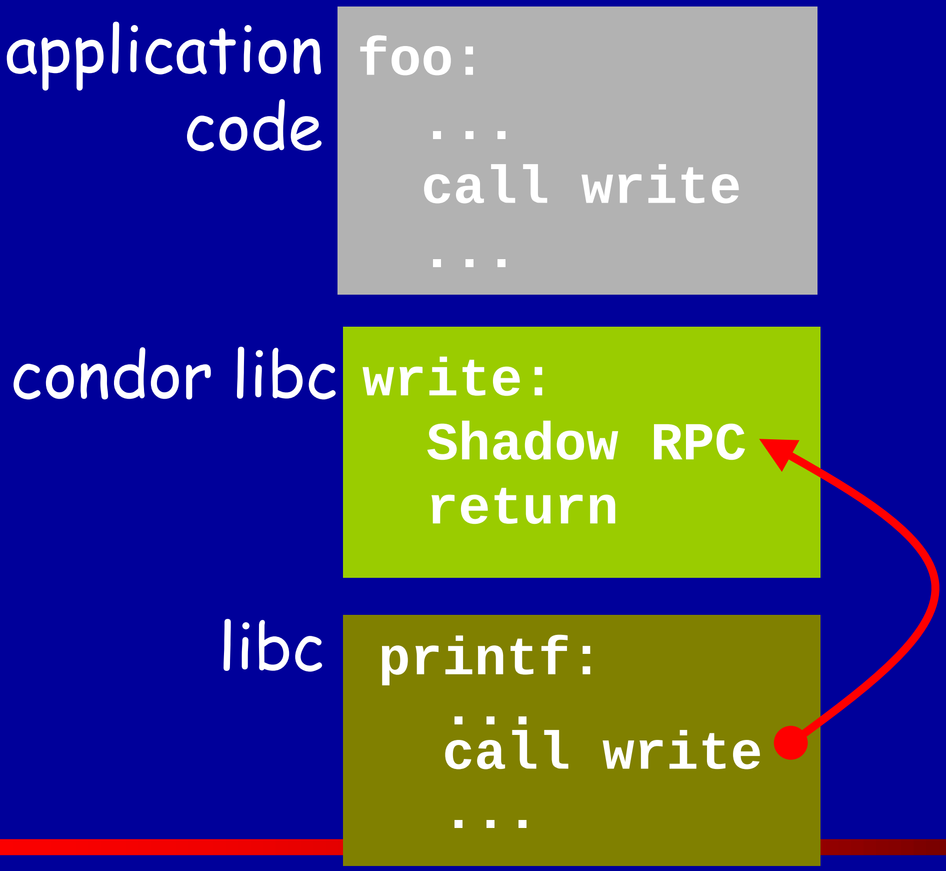
➡ Replace hijacked job's system calls with RPC functions
(replaceFunction)

- Dynamically load RPCs and checkpointing code
(loadLibrary)

- Force the process to fork
(oneTimeCode)

Overriding System Calls

The standard libc remains to support condor libc and other application needs (*e.g.*, **`printf`**, **`malloc`**)



application code

```
foo:
   ...
   call write
   ...
```

condor libc

```
write:
   Shadow RPC
   return
```

libc

```
printf:
   ...
   call write
   ...
```

Costs

Program	Hijack Time	Checkpoint Time (sec)		Restart Time (sec)	
	(sec)	Condor	Hijacker	Condor	Hijacker
tiny	9.0	2.1	3.0	1.6	3.2
big	9.0	12.0	12.8	12.9	14.4
kaffe	10.6	n/a	14.1	n/a	15.6
ss	11.9	3.2	4.4	2.9	4.7

Hijacker checkpoints are 1.5M larger than Condor's

Symbol Counts

tiny	10688
big	10700
kaffe	13901
ss	12374

Counts include all *defined* symbols in the text and runtime loaded libraries

Hijacking introduces approximately 6000 symbols

Hijack Time Breakdown

For ss:

6.7 s	- Parse application text
3.8 s	- Parse injected text
1.0 s	- Replace system calls