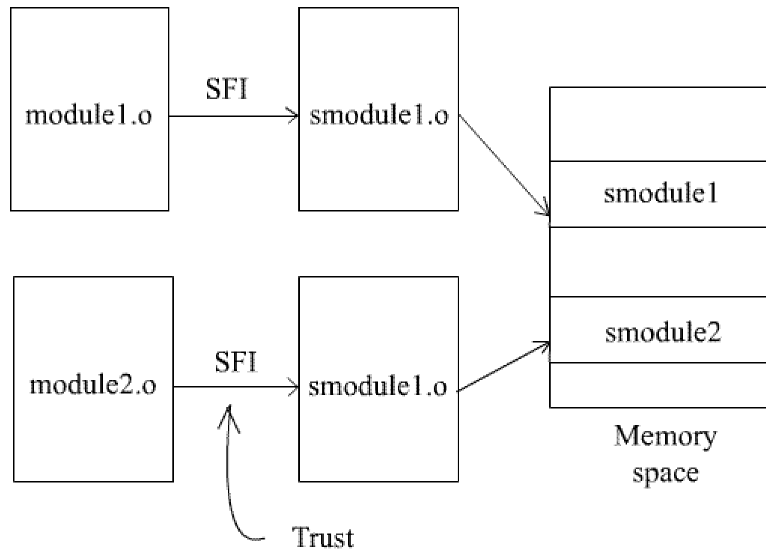
CSE 509 – System Security
Lecture Notes – April 6, 2006

SFI and CFI

To recap, the main idea of software fault isolation is to have two modules that do not trust each other to safely execute in the same address space without reading/writing each other's data and without making jumps to each other's code.



The binaries of the modules are disassembled by the SFI translator and checks are inserted before jump or read/write instructions to restrict the modules to their own domain and the fault-safe modules which are output can safely execute in the same address space.

It should be noted that all the trust lies in the SFI translator, which is relied upon to correctly transform the input modules into safe modules.

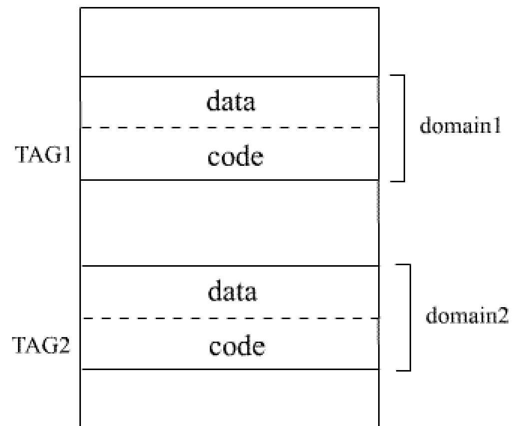
Basic Block Property: Every basic block executes from the beginning.
(A basic block is a sequence of instructions that do not have any jumps. A basic block ends with the appearance of a jump instruction of some kind.)

The primary goal of SFI is to restrict any module to a certain subset of the address space.

We can insert runtime checks to ensure that each module makes read/store calls only to its own domain.

However, code from domain 1 could try to jump into the middle of some code from domain 2. We need to prevent such jumps between domains.

To solve this, we have a unique tag for each domain, and every basic block in any domain begins with a **`label TAGx`** statement, with **`TAGx`** specifying the tag for that domain.



For instance, every basic block in domain 1 would begin with **`label TAG1`** and every basic block in domain 2 would begin with **`label TAG2`**.

Then before every jump, a check could be inserted to make sure that the jump points to a basic block for that particular domain. An example is shown below:

```
cmp eax, TAG1   <- compare target with tag
jne abort       <- abort if mismatch
jmp eax         <- actual jump call
```

There is still a problem. Some code from a domain could try to write to its own code and write some malicious code that it can later execute. Solutions:

- the code can be made read-only
- similar runtime checks can be inserted and it can be ensured that write operations take place only to the data area and not the code area.

We have thus achieved the basic block property and prevented cross-domain read/write/jumps. But we still need to let domains call each other. This is addressed in the section below.

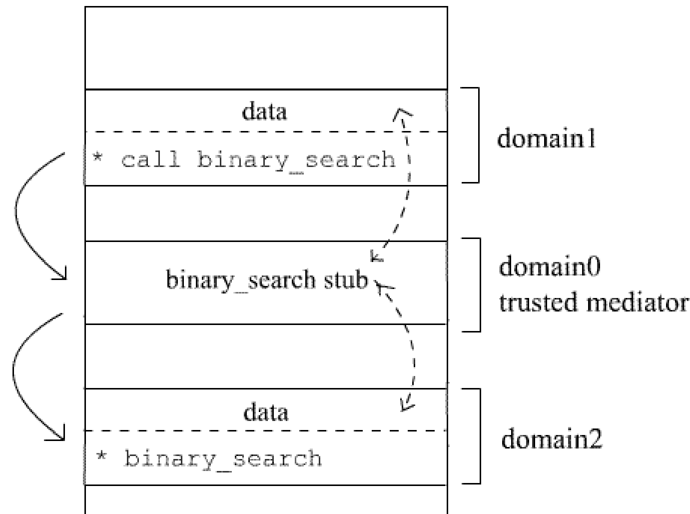
Cross-domain calls via stubs

A stub is a trusted third domain that mediates between two domains that want to call each other.

For instance, let us assume that domain 1 wants to use a binary search function in domain 2. The code in domain 1 actually calls a stub which exists in domain 0, the trusted mediating domain. Domain 0 can read and write data to both domains 1 and 2. It can also jump to a basic block in either domain.

The caller should still be able to jump to the stub in domain 0. Hence, this block in domain 0 is labeled with **`TAG1`** so that code from domain 1 can jump to it. The stub then reads data from domain 1 and passes it to domain 2, and then jumps to the search code in domain 2.

When the processing is done, the code in domain 2 must be able to jump back to domain 0. Hence this block is labeled with **`TAG2`**. All other blocks in the stub are labeled with **`TAG0`** and so neither domain can jump to other code. The trusted domain then reads the results from domain 2's data, writes them to domain 1's data and jumps back to domain 1.



Performance:

In terms of performance, SFI provides low overhead for frequent domain crossings. The relative performance figures are as follows:

Pipe	:	200 microseconds
SFI	:	1 microsecond
Function call	:	0.1 microseconds

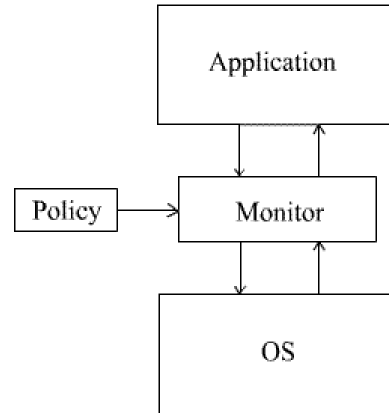
SFI also supports flexible security policies that are independent of the hardware and the operating system.

<u>System Call Interposition</u>

A user may want to restrict the system calls that a downloaded or untrusted application makes. This may take the form of completely blocking certain system calls, or disallowing certain parameters. System call interposition is a method of checking system calls and blocking/changing them if necessary.

System call interposition can be used to configure a user's applications to a restricted access policy. For instance, if a user wants to ensure that a downloaded version of

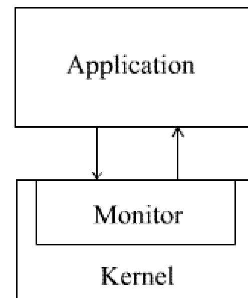
Photoshop that is untrusted does not write files under C:\Windows, system call interposition can be used to check such a condition.



This is generally achieved by placing a monitor program between the application and the kernel. All system calls go through the monitor and the monitor enforces a user-supplied policy.

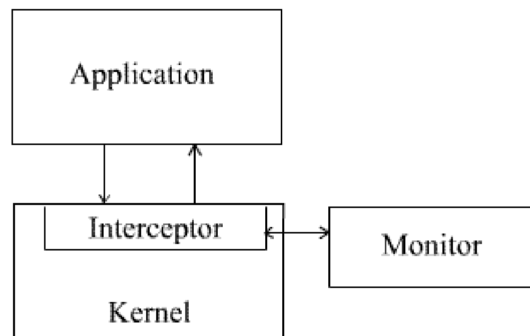
<u>Where to locate the monitor</u>:

A naïve approach would be to place the monitor directly into the kernel at the entry point of system calls. This way, all system calls would go through the monitor which would then pass them to the kernel.



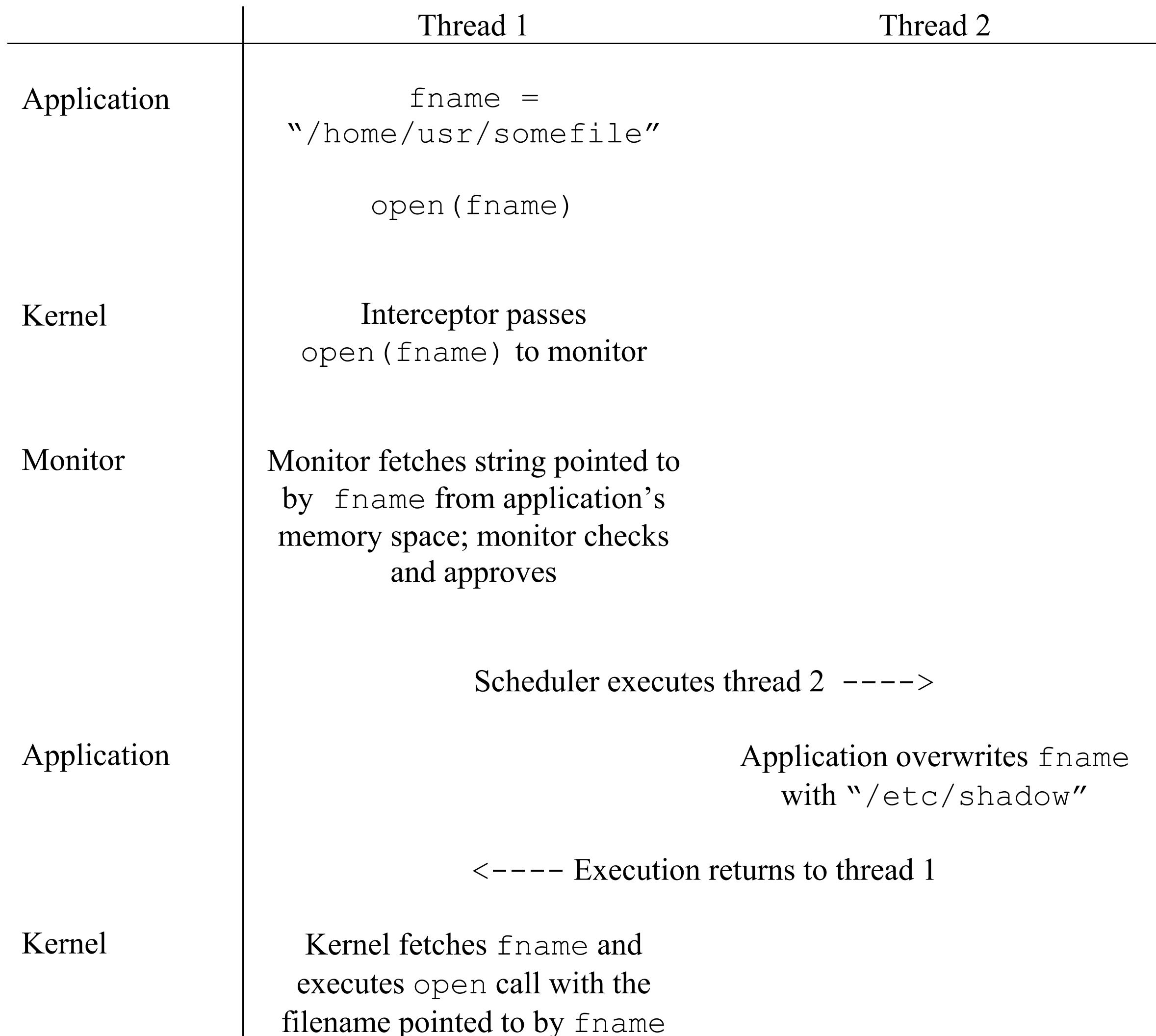
This avoids context-switches during calls, but a monitor may be thousands of lines of code and it can make the kernel bloated.

A better solution is to have a thin interceptor layer within the kernel which intercepts the system calls, and then passes control to the actual monitor which lies outside the kernel.



However, this simplistic method is susceptible to a race condition, as described below:

Consider that the malicious application has multiple threads. Initially, the normal thread of execution is running, and makes a system call to open a file in the user's home directory.



With a little luck, the scheduler will execute the second thread at exactly the right moment, after the monitor has approved the call. The filename will be overwritten with a file that the application should not normally be allowed access to, and when execution returns to the main thread, the kernel will execute the call with the new filename.

* * *