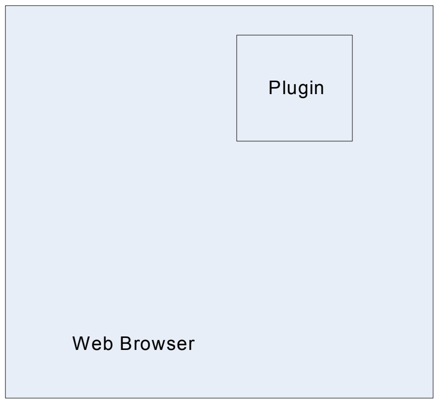
CS 261 Scribe Notes
Sandboxing and SFI
Lecture 7 - 9/18/07

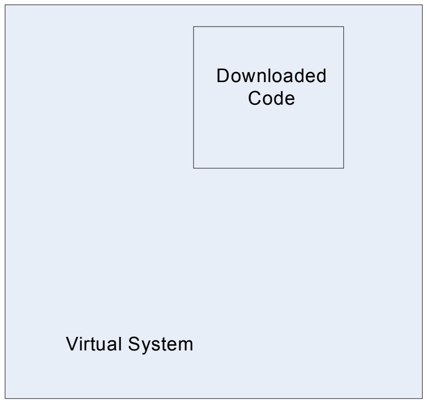
7.1 Sandboxing Examples and Reference Monitors

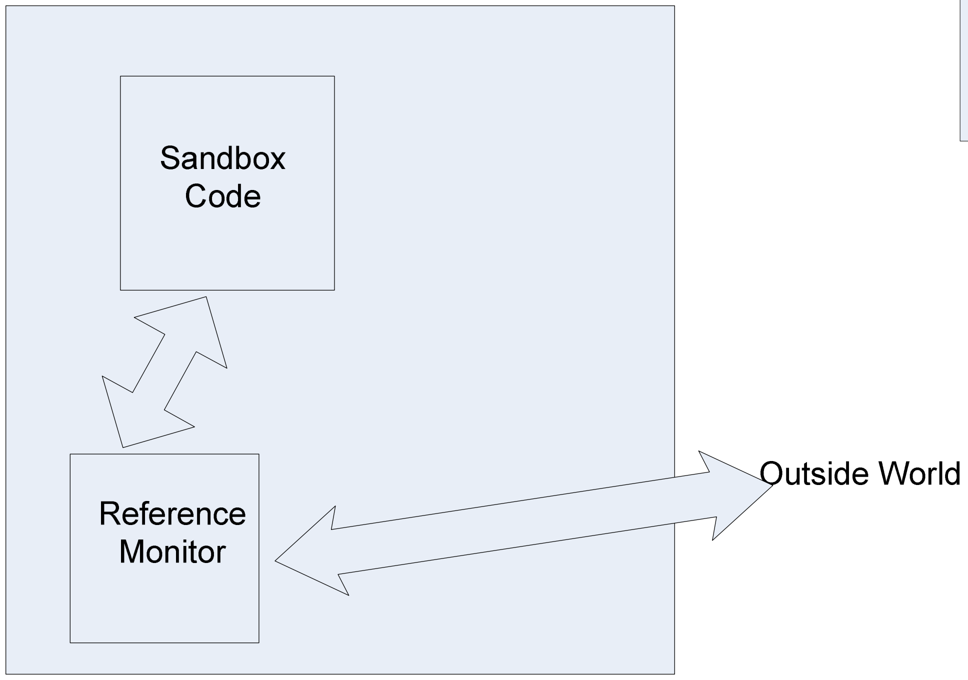
Here are 2 Examples where you might want to use sandboxing:

1) Web Browser with user created and installed plugins.
    a. The plugins are completely untrusted, and you do not want to the plugins to subvert the web browser.
    b. You want to gain some protection against these plugins.



2) Application running in a system.
    a. The application/code is downloaded and is not trusted.
    b. You want to limit this downloaded code from subverting the system in any way.





Code that is not completely trusted communicates with a reference monitor. The reference monitor then communicates with the outside world.

Reference Monitor: If sandbox code wants to interact with anything, it must interact with the reference monitor, which then may interact with the outside world in place of the sandboxed code.

The reference monitor is trying to enforce a security policy on the sandbox code. The policy may be something such as no access to disk or no access to the network. There can be varying security policies with a reference monitor, which must uphold this policy on the sandboxed application.

For example, a user process is the sandboxed code. The reference monitor is the system call that the OS checks permission. The user process can only mess with its own memory except via the OS.

Firefox, for instance, may implement a reference monitor for plugins. The plugins may only be allowed to manipulate certain visual areas of the browser, or access certain browser state, but not the saved passwords in the browser.

7.2 Building a Sandbox

How to build an isolated Sandbox? Splitting allows us to modularize the security into two processes.

You can split this model into two Pieces:

a) Isolated sandbox. The code cannot interact with anything. It is completely isolated.
b) A reference monitor that enforces policy: (User level permission checking code).

SFI is a technique to build this sandbox.
Q: Other ways to build a sandbox?

- Run inside a VM. Run untrusted code in VM. Don't let it access host machine.
  - Just an optimization of buying a 2nd machine and building a point to point network.
- Have a language (such as circuits) that doesn't allow any way to express state such as reading/writing a file.
  - Notion of a **Safe Language**: Impossible to express prohibited actions like writing a file.
  - No side effects in the safe language and must be deterministic – depends only on provided inputs.
- **Hardware Isolation**: Separate OS Process. Rely on OS's memory protection for the sandbox.
- Interpreter: Interprets each instruction and checks if each instruction would violate the isolation policy.

7.3 SFI

Advantages of SFI:

- Is within a process and no marshalling of code is needed. It can just pass a pointer, so it might be faster. (this might be a dubious advantage)
- No context switching since it's the same process.
- More flexible, and has fine grained isolation.
- Don't have to modify the OS. It is very portable. Applications can provide security protection by themselves.
- Smaller Trusted Base since we don't need to rely on the OS.

There may not really be that great of a performance advantage with SFI since it takes a 20% performance hit.

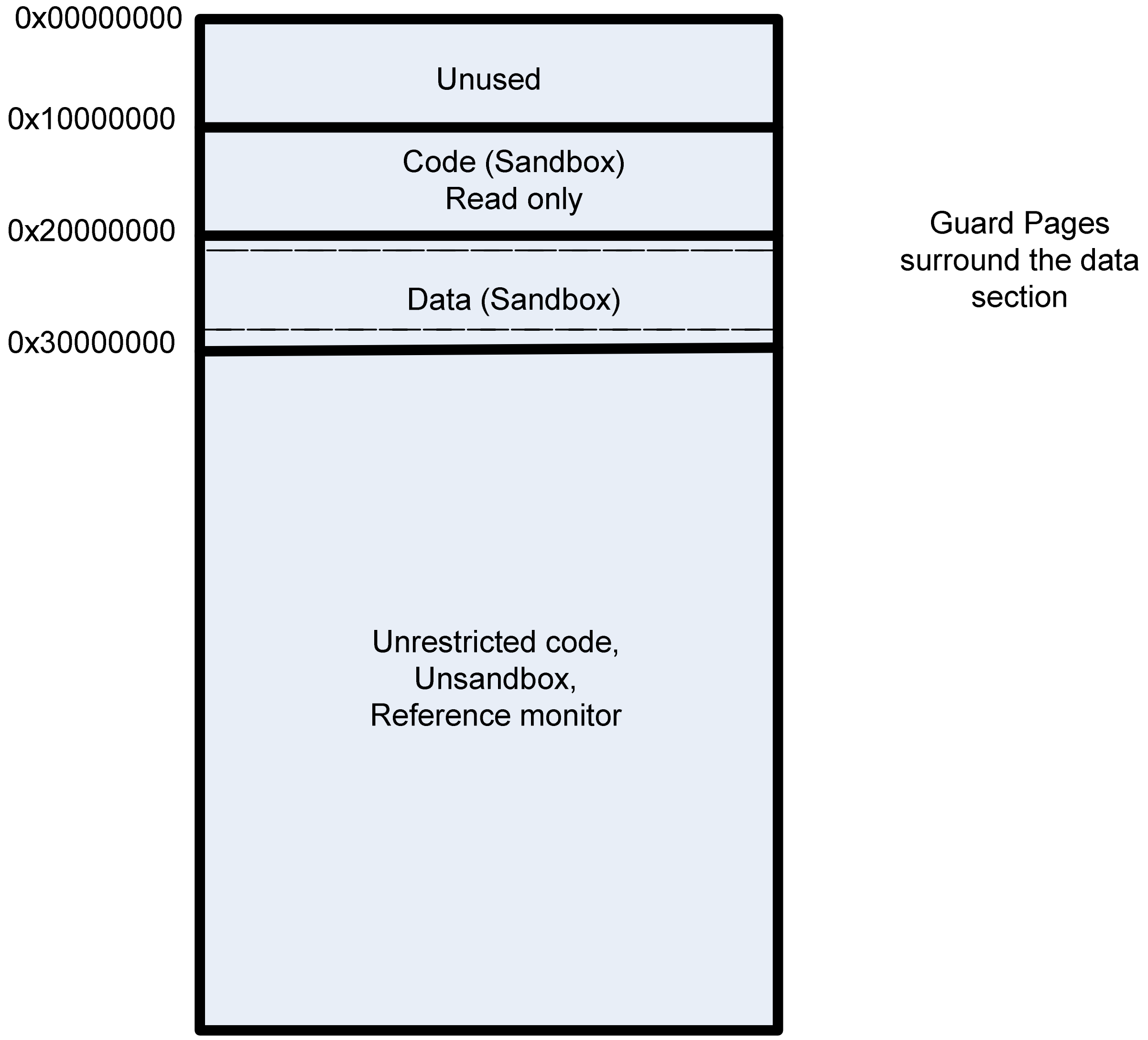
How to do performance comparison? cost of SFI is dependent on application
We need to look at cost per domain call and cost per memory access, as these are the 2 main costs.

Goals of SFI:

Memory Safety: (self modifying code)

Control-flow safety (where code can jump to). Need to ensure sandbox code can only jump to locations within own code segment

Type Safety:



Code = 0x10XXXXXX and Data = 0x20XXXXXX.
Code can only jump to within the 0x10 region, and can only jump on 16 byte chunks so that we can ensure control flow safety.
For memory safety, SFI ensures that code can only write to the data region by restricting the writes to the 0x20 region (inspects the instruction on every store).

We have to know how control/memory safety relate to each other because we need to know where code might jump to, in order to know where data might be written to and vice versa.

mov %eax, %(ebx) gets translated to:

and 0x020FFFFFF, %ebx
mov %eax, %(ebx)
This is to ensure that every move/store, the data stays within data region. Since there may be offsets to the %ebx register, guard pages are used around the data region (that is unmapped) so that you cannot write outside the data region. This way, only the register needs to be checked via the and instruction. We do not have to check the offset in the mov instruction.

Check all jump instructions

a. Jump 0x10______ <- doesn't need to change as long as target is there and okay and aligned in the 16 byte chunk. We needed to prevent jumping to other artificially introduced chunks because instructions are of variable length. If we do not jump to an aligned chunk, we may be able to jump to a 'hidden' set of instructions if we do not really jump to the beginning of an instruction. Before this idea, SFI could only be RISC because of the variable length instructions. (NOPS are inserted in instruction regions to make sure certain instructions are aligned on the proper chunks).
b. Indirect branch
   Jump %ebx gets translated to:
   and 0x10FFFFFF, %ebx
   Jump %ebx
   This ensures that you can only jump to the 0x10 region or the 0x00 region. You must un map 0x00 region or security is compromised since data can also write to 0x00.

How would you extend SFI to use a reference monitor?

We can allow key jumps to predefined location at specified entry points in unrestricted code. We force this to only work for direct jumps, because indirect jumps are messy. Indirect jumps are difficult because there may be offsets for the register, so we can just disallow indirect jumps into the unrestricted code. This moves the burden onto the code writer.

We can't allow reference monitor to jump back to code in unaligned spot, so we must use check the return address to jump back to the proper aligned location in the 0x10 code region. Also, the malicious sandbox code can give us a bad return address, so we must use jump checks on the return address.

The reference monitor is on the same stack being used as the untrusted code, so in order to prevent data from being overwritten by malicious code in the sandbox, the reference monitor must switch to using its own private stack.

There is also no guarantee that malicious code uses the proper callback structure. The reference monitor needs to save and restore their own data on the stack instead of relying on the code to adhere to the callback structure. .

<u>7.4 Analogy on Isolation</u>

OS	**SFI**
Kernel	reference monitor
User process	sandbox code
System call	invoking of reference monitor

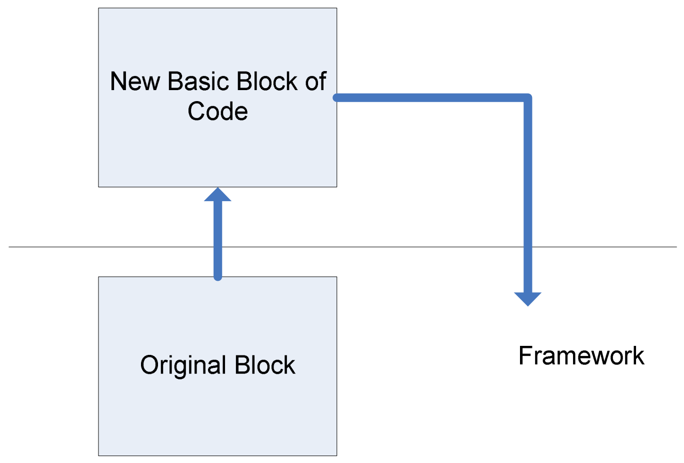
Comparison of security guarantees by SFI and the OS

- Confidentiality: SFI lets you read entire address space, as there is no sandbox on reads in SFI.
- Availability: SFI doesn't prevent malicious code from using DOS attacks (infinite loops). In order to fix this, we can have a reference monitor that starts a timer before starting malicious code. When the timer exception occurs, jump back to reference monitor code to determine if the code in the sandbox can still run.

7.5 Techniques for binary transformation

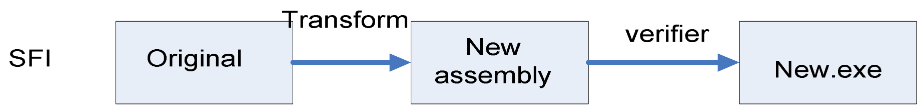
1) Source/Assembly -> binary code (statically) (SFI)
2) Binary code-> binary code (runtime on the fly) Dynamic (RIO, VMWare, Plex86)
3) Binary code -> binary code (statically)
   This is in order of difficulty to implement, and 3) is a real pain to do.

Number 2 is implemented by having an infrastructure that simulates x86 at high level, The general process is, transform the original block of binary code, execute this new block of binary code, at the end jumps back to the runtime transformation framework. This framework then translates the next block of binary code and so forth.



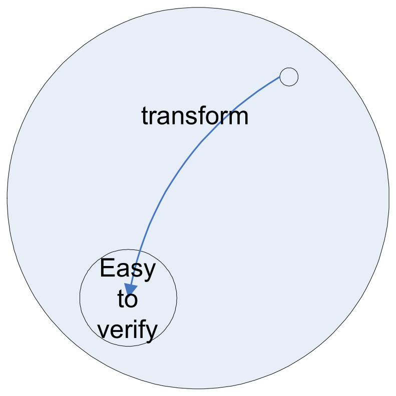
General implementation of transforming binary code->binary code during runtime.

The SFI process translates assembly code to binary code statically.



After the transformation, the verifier checks to make sure that the new assembly is secure. This allows us to leave the transformation code, which may be very complicated, outside the TCB. Only the small verifier lies in the TCB.

The notion of "transform and check" is popular in security. It is good to look for opportunities where we can transform code into an easier sub case that we can verify is safe and secure.


transform
Easy to verify