Address Space and Virtual Memory (Software Fault Isolation)

Dai Wentao, Wang Zixuan

why we need fault isolation?

Imaging you added some mods in minecraft

You’ d think some mods are harmless,
but then you suddenly discover
all your saves are gone for no reason!

Subconsciously, you’ ll find yourself
blaming the game itself
rather than the mods you installed.

why we need "Software-Based"fault isolation?

back to 1990s...

memory: 4MB

expensive context switch

UNIX®
A Standard of The Open Group®

🔥popular

java? 1995 not popular

Java™

VMware? Founded in 1998
full virtualization heavy

vmware

Scenario	Overhead	Main Reason
Function Call	~0.1 μs	Only a few registers are saved; no kernel intervention.
SFI Cross-domain Call	~1 μs	Involves a user-mode stub; saves/restores registers.
Thread Switch	1-5 μs	Kernel intervention; saves/restores registers; no page table switch.
Process Switch	5-20 μs	Kernel intervention; saves/restores registers; switches page table and TLB.

why we need "Software-Based"fault isolation?

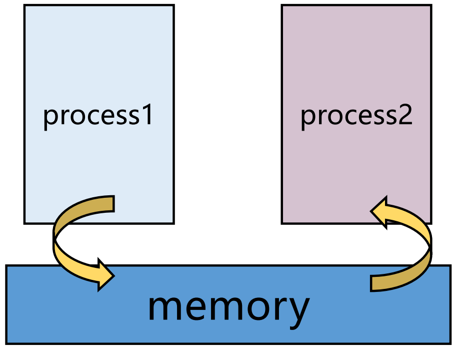
Distributed System

limited computing resource

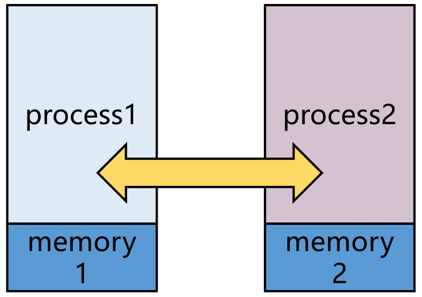
increasing demand of IPC (the development of database,PostgreSQL: 1995， MySQL:1995)

IPC

shared one memory



isolating memory(hardware isolation)



how to communicate from p1 to p2

cooperation through memory replication?

RPC(remote procedure call)

a protocol that allows a program to request a service from a program on
a different computer on a network without having to understand the network's details

note: here we talk about cross-domain RPC which means they can share memory

Back to the MULTICS design philosophy

multics

Establishing Sharing:

- Process A stores the "Paragraph" data in a separate segment (paragraph_seg).
- A makes a system call, instructing the OS to grant Process B read access to paragraph_seg.
- The OS then links paragraph_seg into Process B's address space, allowing both processes to see the same segment.

Communication & Processing:

- A notifies B that the "data is ready" using a lightweight sync mechanism (e.g., a shared semaphore segment).
- Once awakened, B directly accesses paragraph_seg from its own address space, seeing A's original data without any copying.
- B reads the data, performs the spell check, and writes the result into another shared segment (result_seg).
- B notifies A via the sync mechanism, and A reads the result directly from result_seg.

Back to the MULTICS design philosophy

unix

Establishing a Channel
A and B create a communication channel (e.g., pipe, socket) via a system call.

A Sends Data

- Copy 1: A serializes the "Paragraph" data into a buffer.
- Syscall: A executes write(), trapping into the kernel.
- Copy 2: The kernel copies the data to its own buffer.
- Schedule: The kernel wakes up the waiting Process B.

B Receives Data

- Syscall: B executes read(), trapping into the kernel.
- Copy 3: The kernel copies the data to B's user-space buffer.
- Return: B now has its own copy of the data.

too slow?

Return Path
B processes the data and returns the result, repeating the entire multi-copy process in reverse.

Back to the MULTICS design philosophy

unix

1. Setup & Mapping

A creates & sizes a shared memory object (my_document_share) via shm_open() and ftruncate().

Both A and B map the same object into their respective address spaces using mmap(). Their page tables now point to the same physical RAM.

A and B create two semaphores (data_ready_for_b, result_ready_for_a) for synchronization.

2. Zero-Copy Data Flow

A → B:
A writes data directly to its mapped address → A posts data_ready_for_b → B waits on & acquires the semaphore → B reads data directly from its mapped address.

B → A:
B writes results directly to the shared area → B posts result_ready_for_a → A waits on & acquires the semaphore → A reads results directly.

SFI said: not fast enough

what SFI do ?

1. Sandboxed Loading

    At startup, Editor A loads spell checker B's library.

    The SFI loader rewrites B's code, confining it to a dedicated "Fault Domain" (e.g., 0xBEEF0000-0xBEEFFFFF).

    It inserts guards into B's instructions to enforce domain boundaries and creates a secure Jump Stub for B's check_spelling function.

2. Cross-Domain Call

    A's Call: A prepares data and calls check_spelling(paragraph_data, result_buffer). This jumps to the secure stub.

    Context Switch: The stub configures SFI registers (e.g., base address 0xBEEF0000) and jumps to the sandboxed check_spelling.

3. Execution & Isolation

    B's Work: The sandboxed check_spelling executes:

    - Reads paragraph_data directly from A's memory (reads are typically permitted).

    - Computes within its own fault domain.

    - Writes the result to A's result_buffer (writes to authorized caller buffers are permitted; unauthorized writes are trapped/redirected).

Note: Here, A and B are no longer separate processes but two modules within a single process.

4. Return

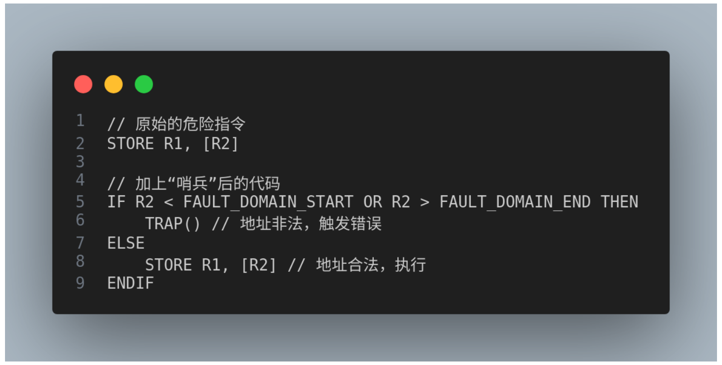
    B returns. The stub cleans up SFI context and returns control to A.

There's not even a context switch.

How to implement

SFI has a clear objective: to implement strict isolation within a single process through software alone, without compromising performance. Let's break down its construction.

simple ideal:We can place a "sentry"（哨兵）
—a segment of inspection code—before every potentially dangerous instruction.



```
// 原始的危险指令
STORE R1, [R2]

// 加上“哨兵”后的代码
IF R2 < FAULT_DOMAIN_START OR R2 > FAULT_DOMAIN_END THEN
    TRAP() // 地址非法，触发错误
ELSE
    STORE R1, [R2] // 地址合法，执行
ENDIF
```

what is dangerous instruction?

Memory Stores (e.g., STORE R1, [R2]): Potentially corrupt data by writing outside the allowed domain.

Indirect Jumps (e.g., JUMP [R3]): Could transfer control to an invalid address, bypassing intended function entry points.

However, this method introduces a prohibitive performance cost.

Challenge 1: Must we check every hazardous instruction?

The SFI authors observed that many "dangerous" instructions can be statically verified as safe at compile time, eliminating the need for runtime checks:

Static Variable Writes: STORE R1, global_variable is safe if the compiler knows global_variable resides within the module's data segment.

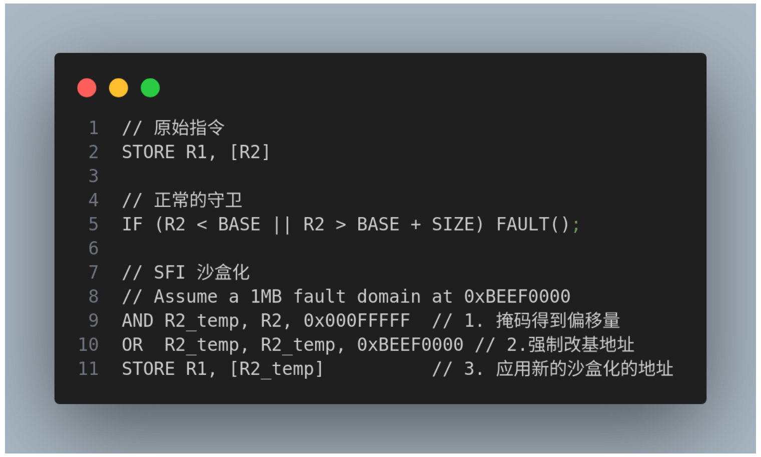
Stack Writes: STORE R1, [StackPointer + offset] is safe as long as the stack pointer is controlled.

PC-Relative Branches: BRANCH label is safe if the target is within the module's code segment.

Challenge 2: The performance overhead of checks remains.

Two fast, fixed bitwise operations (AND, OR) replace slower compare-and-branch instructions. This guarantees the address (R2_temp) always falls within [0xBEEF0000, 0xBEEFFFFF].

This method is both perfectly safe and extremely fast. The minimal overhead of these bitwise operations on modern CPUs is the key secret that allows SFI to achieve remarkably low performance overhead (around 4%).

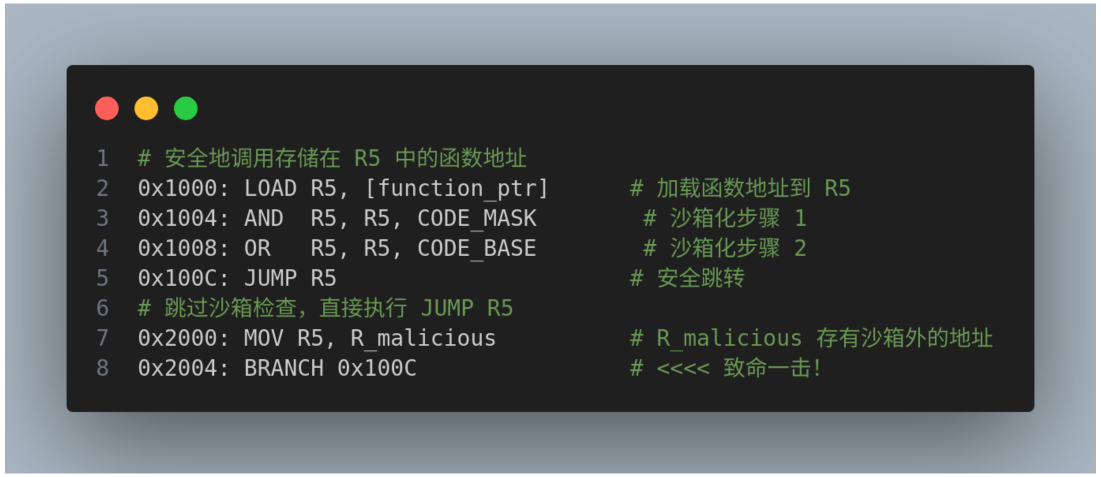


```
// 原始指令
STORE R1, [R2]

// 正常的守卫
IF (R2 < BASE || R2 > BASE + SIZE) FAULT();

// SFI 沙盒化
// Assume a 1MB fault domain at 0xBEEF0000
AND R2_temp, R2, 0x000FFFFF  // 1. 掩码得到偏移量
OR  R2_temp, R2_temp, 0xBEEF0000 // 2.强制改基地址
STORE R1, [R2_temp]          // 3. 应用新的沙盒化的地址
```

safe and fast?

addtional challenge

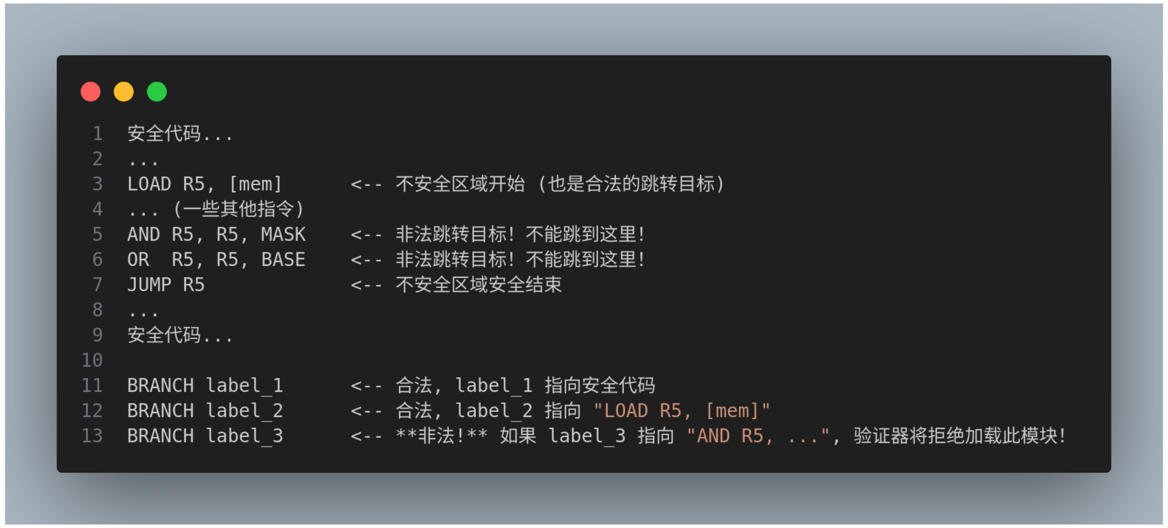
An attacker might try to bypass the sandboxing code by crafting R5 to jump directly to the final JUMP R5_temp instruction.



```
# 安全地调用存储在 R5 中的函数地址
0x1000: LOAD R5, [function_ptr]       # 加载函数地址到 R5
0x1004: AND  R5, R5, CODE_MASK         # 沙箱化步骤 1
0x1008: OR   R5, R5, CODE_BASE         # 沙箱化步骤 2
0x100C: JUMP R5                       # 安全跳转
# 跳过沙箱检查, 直接执行 JUMP R5
0x2000: MOV R5, R_malicious           # R_malicious 存有沙箱外的地址
0x2004: BRANCH 0x100C                 # <<<< 致命一击!
```

addtional challenge

Guaranteeing Sandbox Code Atomicity: The verifier inspects every "unsafe region." It must ensure that from the start to the end of the region, there exists no other control-flow path that can jump into the middle of it. In other words, once entry into an unsafe region occurs, execution must proceed sequentially all the way through until the jump has been safely sandboxed.



```
安全代码...
...
LOAD R5, [mem]      <-- 不安全区域开始 (也是合法的跳转目标)
... (一些其他指令)
AND R5, R5, MASK    <-- 非法跳转目标! 不能跳到这里!
OR  R5, R5, BASE    <-- 非法跳转目标! 不能跳到这里!
JUMP R5             <-- 不安全区域安全结束
...
安全代码...

BRANCH label_1      <-- 合法, label_1 指向安全代码
BRANCH label_2      <-- 合法, label_2 指向 "LOAD R5, [mem]"
BRANCH label_3      <-- **非法!** 如果 label_3 指向 "AND R5, ...", 验证器将拒绝加载此模块!
```

Cross-Domain RPC

How can untrusted code call external services (e.g., to print a log), and how can trusted code call functions inside the sandbox?

Exiting the Sandbox: The Jump Table

Sandboxed code cannot directly execute system calls or arbitrarily call external functions.
SFI provides each sandbox with a unique, read-only Jump Table. This table is essentially an array of addresses pointing to the entry points of all authorized external "safe" functions (e.g., safe_print, safe_malloc).

To call an external service, sandboxed code must perform an indirect jump through this table (e.g., JUMP [JUMP_TABLE + index]). This is safe because the table itself is read-only and resides in trusted memory.

Entering the Sandbox: Secure Stubs

As discussed, external code must call sandboxed functions through an SFI-generated Secure Stub.
Beyond setting up dedicated registers, this stub is responsible for parameter marshaling and return value handling, ensuring safe and expected data exchange across the domain boundary.

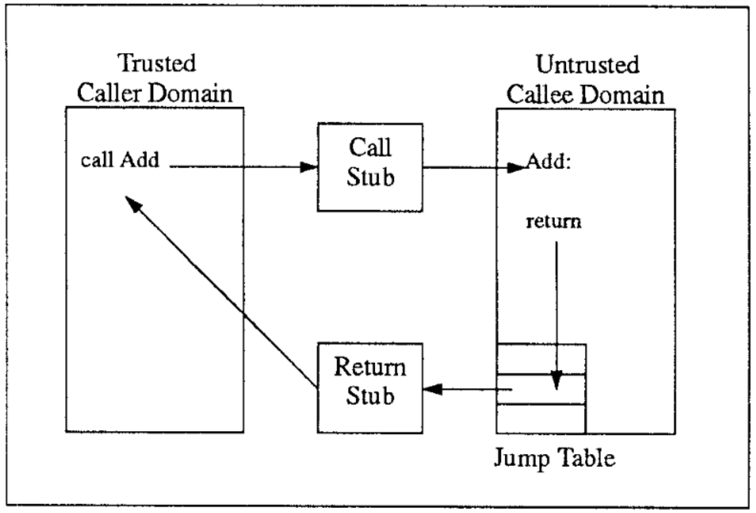
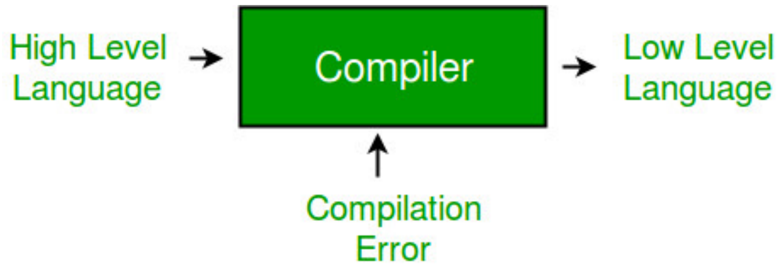


Figure 4: Major components of a cross-fault-domain

When to implement SFI?

Strategy 1: Compiler-based Approach



Used in this paper
Relatively Straightforward
But,protecting Rust or Go
modules would require modifying
their respective compilers.

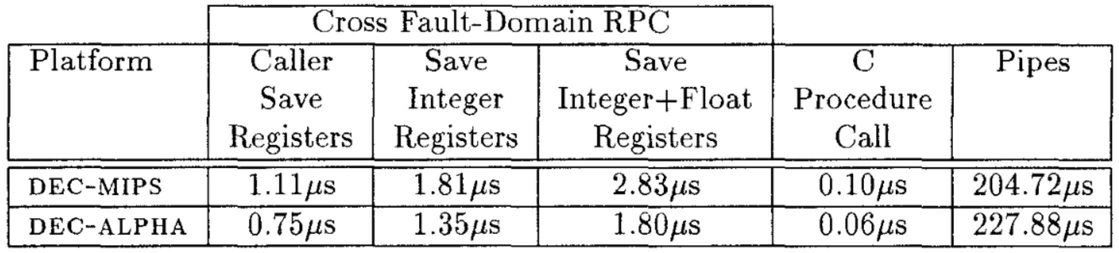
Strategy 2: Binary Patching Approach

complex ,not implemented yet?
But this ideal futher affect Google NaCl

In a complex application, multiple sandboxes may exist. For instance, a web server might create a separate fault domain for each plugin. How can these plugins share data securely?

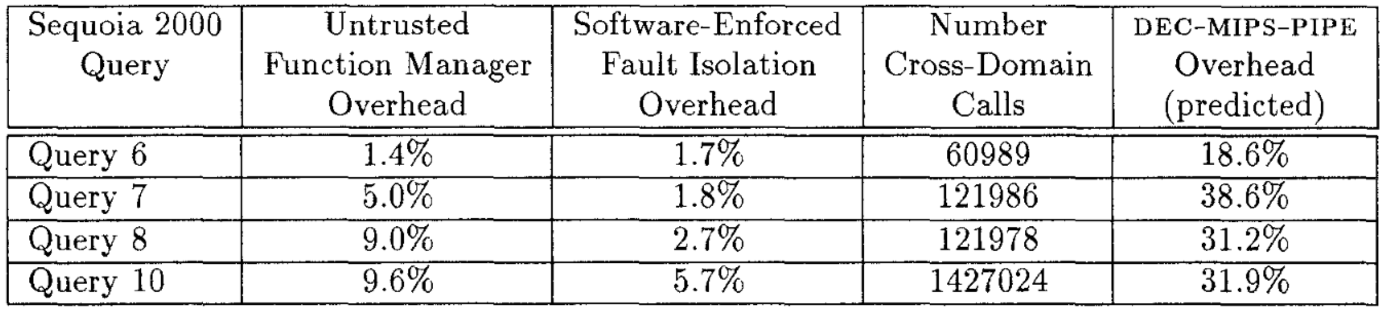
- Read-Only Sharing: This is straightforward. Since SFI does not restrict read operations by default, any sandbox can read the entire address space.
- Read-Write Sharing: This is more complex. The SFI paper proposes several schemes:
    - Lazy Pointer Swizzling: This technique leverages the OS's virtual memory subsystem to map the same physical memory at the same offset within different sandbox domains. Although the "high bits" (segment base) of pointers differ, the "low bits" (intra-segment offset) are identical. The built-in address sandboxing logic then automatically "translates" any pointer to the correct address within the sandbox's own domain. This enables zero-copy, high-performance read-write sharing.
    - Shared Segment Matching: If the OS does not support such memory aliasing, a more general software scheme can be used. A dedicated register maintains a bitmap indicating which other segments the current sandbox is authorized to write to. Before each write, in addition to address sandboxing, an extra instruction checks this bitmap for permission.

Why faster?



Platform	Cross Fault-Domain RPC Caller Save Registers	Save Integer Registers	Save Integer+Float Registers	C Procedure Call	Pipes
DEC-MIPS	1.11μs	1.81μs	2.83μs	0.10μs	204.72μs
DEC-ALPHA	0.75μs	1.35μs	1.80μs	0.06μs	227.88μs

Table 2: Cross-fault-domain crossing times.



Sequoia 2000 Query	Untrusted Function Manager Overhead	Software-Enforced Fault Isolation Overhead	Number Cross-Domain Calls	DEC-MIPS-PIPE Overhead (predicted)
Query 6	1.4%	1.7%	60989	18.6%
Query 7	5.0%	1.8%	121986	38.6%
Query 8	9.0%	2.7%	121978	31.2%
Query 10	9.6%	5.7%	1427024	31.9%

Table 3: Fault isolation overhead for POSTGRES running Sequoia 2000 benchmark.

When to use SFI

savings = (1− r)t_c−ht_d

tc: the percentage of time spent crossing among fault domains

td: the percentage of time spent in distrusted code

h:the overhead of encapsulation

r:ratio of fault domain crossing time to the crossing time of the competing hardware-based RPC mecha-nism

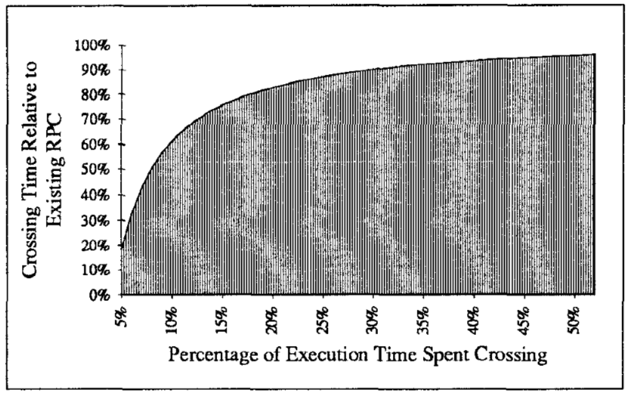


Figure 5: The shaded region represents when software enforced fault isolation provides the better performance alternative. The X axis represents percentage of time spent crossing among fault domains (t_c). The Y axis represents the relative RPC crossing speed (r). The curve represents the break even point: $(1-r)t_c = ht_d$. In this graph, $h = 0.043$ (encapsulation overhead on the DEC-MIPS and DEC-ALPHA).

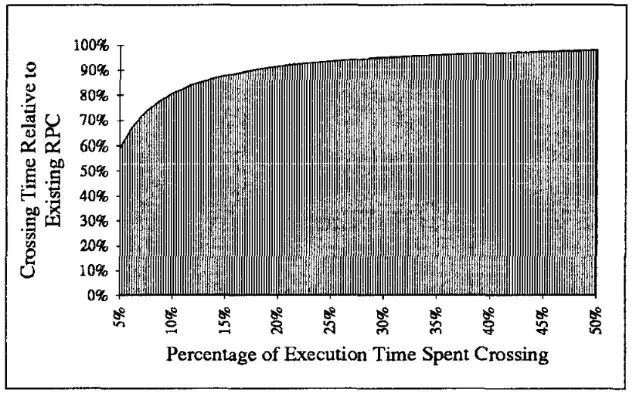
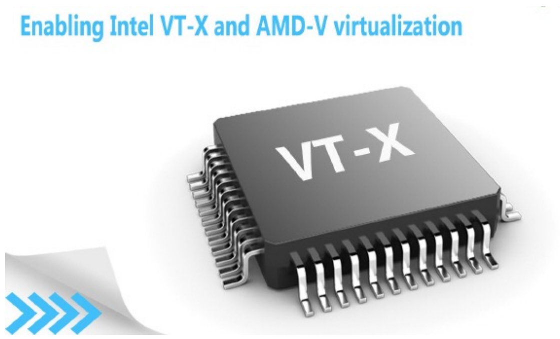


Figure 6: The shaded region represents when software enforced fault isolation provides the better performance alternative. The X axis represents percentage of time spent crossing among fault domains (t_c). The Y axis represents the relative RPC crossing speed (r). The curve represents the break even point: $(1-r)t_c = ht_d$. In this graph, $h = 0.043$ (encapsulation overhead on the DEC-MIPS and DEC-ALPHA).

SFI died?

Proliferation of Virtualization



Safe Programming Languages & Runtimes

.NET JVM Java Virtual Machine eBPF

SFI died?

WA
WEBASSEMBLY

From the hardware utopianism of Multics, through the software realism of SFI, to the ubiquitous sandboxing in today's browsers, operating systems, and safe languages, we are witnessing a grand interplay between security and performance that has spanned half a century. Standing in this long river of history, SFI serves as a pivotal bridge, connecting the past with the future. Its core ideas will continue to be a fundamental source of inspiration for building the next generation of secure and efficient software systems.

Thanks!