Inline Reference Monitors

Michael McCoyd

7 Septemeber, 2012

1 SFI sandboxing

SFI is a way to mitigate risk of untrusted guest code by constraining what it can do. For example running a image decoder or arbitrary plugin in a browser at close to full speed. Sand Box " some part of address space for guest" can not trample on browsers (host) data structures.

SFI on RISC Can modify the code so nothing bad can happen. All instructions are fixed length so for a given byte stream there is only one instruction stream.

Why is SFI on CISC hard SFI on CISC was harder as variable length instructions. The attack can jump to an address in the middle of an intended instruction and thus create a separate instruction stream. Variable length instructions makes it impractical to modify code to always be safe.

PittSFIeld: transform CISC into fixed length "instruction" architecture Was the next revolution and showed could have fast SFI on x86. Approach was to ensure that jumps only occur to the start of 32bit chunks, by masking addresses.

32 bytes Need to fit in chunk all the checks you want for that instruction, else you could jump past the checks. 32 bytes gives enough room if rewrite any really long x86 instructions. Also a power of 2 given bit masking method of enforcing addresses.

2 SFI Development Path

SFI (RISC) -> PittSFIeld (CISC) -> (5 years) -> NaCL x86 w/ segments -> NaCL x86-64, ARM

Native Client x86 32bit (NaCl) key: use x86 segment registers to reduce overhead of PittSFIeld. x86 has two memory protection mechanisms: segmentation and page tables. 64 bit removed segmentation as not often used. Criticized as "locking" web into 32bit and x86, not ARM as well.

NaCL ARM and 64bit x86-64 and ARM with modest success, but only modest. 20 years deployment from initial RISC SFI to cross platform SFI.

Inputs Pitsfield needs assembly code (source), research system, can not be applied to arbitrary binaries. NaCl - modified complier - need the source to instrument the binary.

2.1 Trusted Computing Base

A fundamental part of PittSFIeld is that rewriter need not be trusted, only the verifier.

Trusted Computing Base (TCB) code that needs be correct for security properties to be true.

Instrumentor takes x86 program source and produces an instrumented x86 program. Add masks and checks before jumps. -> instrumented Not part of the TCB. Bugs in instrumentor could only cause it to fail to accept program or change program semantics.

Verifier at runtime verifies properly instrumented. Checks each jump or store immediately preceded by mask instruction.

The security property that we want is sandboxing: not able to write outside its designated security area. For PitSFIeld, the TCB is only the verifier (and the Browser/OS). The PitSFIeld TCB was only intended to assure security, not proper execution. Crucially this split architecture allows the verifier to be much simpler than the rewriter.

2.2 Why was this separation possible?

1) Transform code to subset of x86 that was easier to verify Change a hairy instruction set into a simpler subset that is easier to verify. For example, change crazy address modes to easier code. Instead of: "eax + 8 * ebx + 17 ", use lea to figure out what address that is, load into other register ecx, then mask ecx, then store to address in ecx.

2) Sufficient conditions, but stricter than necessary The transformation and verification are sufficient to ensure we do not write outside sandbox but may be more strict than needed to be. The verifier can be simpler as it has no memory across chunks. Verifier assumes nothing about a block of code at its start. Marks registers as safe for memory or jumps as their values are masked. Takes about 2 bits for each register at each point in code; 1 bit for code, 1 bit for data.

Transform and check In important technique is to map a hairy problem (in this case an instruction set) into much smaller subset with easier semantics. Here it is easy to see that the smaller subset is safe and a verifier for it is possible. The insight needed is in picking the safer subset or problem. A problem with the technique is that it can reduce efficiency.

Hardware requirements PitSFIeld does not require NOX bit, nor use if hardware has it.

2.3 PittSFIeld Memory Design

PittSFIeld had clever memory layout so memory checks are just masking.

Base address	Use
0x30000000	host code & data
	guard pages
0x20000000	guest data & stack
	guard pages
0x10000000	guest code
	guard pages
0x00000000	Zero tag region

The sandbox is the guest code and data areas, maybe also the zero tag region.

Invariant: The guest code will never write outside the guest data region
How: instrument every write instruction
Need to not only prevent writes outside of guest area, but also writes that could change the code and remove its write protections.
Ensure: every write instruction in code is what was loaded in
So: need to protect code are from writing to prevent changes to the instructions.
Method: every write instruction is masked

2.4 Examples

2.5 Write memory

Original	Rewritten	Comments
mov eax, (ebx)	and 0x20FFFFFFFF, ebx mov eax, (ebx)	mask ebx to either 0x00..... (unused zero tag) or 0x20..... (guest data)

Could imagine check with branch, but branch instructions are slow. It is faster to just force it into that range. Writing to the zero tag area is fine as either never use that area or set the OS to mark them not writable.

2.6 Jump

Original	Rewritten	Comments
jmp 0x12345678	[no change]	Constant address so statically reject or allow
jmp (eax)	AND 0x10FFFFE0, eax jmp (eax)	0x10... masks to guest code area 0x....E0 alignes to start of chunk

The masking of a jump will allow execution to enter the zero tag region. Thus that region needs to be marked not used by the page tables.

2.7 Return

Return is effectively an indirect jump, which is unsafe.

Original	Rewritten	Comments
return	pop eax and 0x10FFFFE0, eax jmp (eax)	Return address on stack in guest data area, function body could have changed the return address.

Problematic for performance as modern x86 hardware caches recent return addresses and will speculate where an upcoming return will go. This rewrite removes the return speculation. A significant performance overhead for programs with many call returns.

Concurrency A return optimization that does not work is to AND the return addess in place on the stack just before the return. This preserves the speculation gains. It does not work in the presence of concurrency. With two guest threads one thread could change the return address between the mask and the return of the other. Concurrency was not an issue with the earlier examples as registers are thread local, saved and restored between threads.

Can extra hardware support solve the problem? Instruction that blocks other threads for one instruction. Might have different threads of same process have different protection views of the same memory.

How generally solve concurrency issues? Common defense against malicious code changing your data is to make private copy in local state know others can not modify. Easy to erroneously write: "check(x), do with(x)". **"time of check to time of use"** vulnearability, a race condition on shared mutable state. This is why kernels turn off all concurrency for critical actions.

2.8 Read memory

Original	Rewritten	Comments
read (ebx), eax	[no change]	Reading is not part of their security claims. User must judge if PittSFIeld's claims met users needs.

Does allow leak of privacy. Fixing by checking all reads is big performance hit.

2.9 SFI vs OS process separation

Why do we need both SFI and OS processes? What are the tradeoffs of choosing between them? Seems a lot of similarity in wanting to ensure isolation between the guests/processes. Using an example of a video player, how do we communicate encoded video to the guest?

Standard argument, but no great performance comparisons, is:

Feature	SFI (PittSFIeld)	OS
Call speed host to guest	fast - no need to copy data	slower - fork, copy data
Guest to host writes	Must break sandbox	Same context switch as call.

2.10 Details

Feature	SFI (PittSFIeld)	OS
Basic setup	Code rewriting for separation	page tables and memory protection to ensure each has own virtual address space.
Startup costs	compile time rewrite	forking overhead
share data - maybe SFI faster	**host→guest** just write data into guest, read any guest data. **host←guest: read** just access, or browser store originally in players area. **host←guest: write** ???	IPC with API you create
Multiple guests	No - memory layout prevents	Yes
Memory mapped files	Limited	up to address space size
Cross domain calls	host→guest: trivial host←guest:	IPC marshal
Program changes	Normal access when reading data structures in host	Need to marshal / unmarshal for each access
access to resources	rewrite	??

Other alternatives to SFI are lightweight mechanisms to run in virtual machine. OSDI 2012 had an interesting paper "Dune: Safe User-level Access to Privileged CPU Features" that used hardware virtualization to give a process limited access to CPU privileged features.

Clearly SFI is useful for very specialized low end hardware with out memory protection.

2.11 Can Guest talk to Host?

How does SFI return any results to the caller, as masked inside sandbox.

Imagine running a guest video encoder. Version 1: only write to from buffer on stream. That is nice but also want to connect back to wed site for streaming video. Lets say that is ok to the site we came from but not other sites.

How do we do that?

Adding access code into the guest will not let that code talk to the browser, as still sandboxed. Could store arguments in guest data and cause host to read and execute on them. We could use page faults or other exceptions to transfer to host, but we are reimplementing system calls.

Cleaner is to change verifier to allow one address that you can call to outside the sandbox, a clean implementation of system call.

Stack Where is stack in this process? Guest and host would share a stack. So one guest could read the stack after host use. If two guests were running, one could modify the hosts stack out from under it. If such concurrency, host copies arguments to a separate stack used just by the host. If host call back to guest, host should save its registers before call guest.

Make lots of defensive copies with mutual distrusting parties in face of concurrency, much like you need in OS design. Paper did not discuss. The interesting issues of OS design show up in SFI as well. Could allow no callbacks to guest.

3 Reference Monitor Pattern

Given sandboxed guest code with no external access and host code can do anything such as accessing the network. You want to allow the guest a few limited abilities, say 10, such as to connect back to its website.

The pattern is to use a Reference Monitor. The guest makes its request to some code within the host, called the reference monitor. The reference monitor checks if the guest is allowed to do the requested action. Instead of changing the SFI verifier to allow all 10 things.

Decompose problem into two pieces. SFI verifier that completely sandboxes guest except one entry point to the host. Reference monitor in host checks those 10 things. Similar to system calls.

4 SFI limitations summary

- only one guest
- return gets slower (fix with mask and memory)
- privacy not a goal