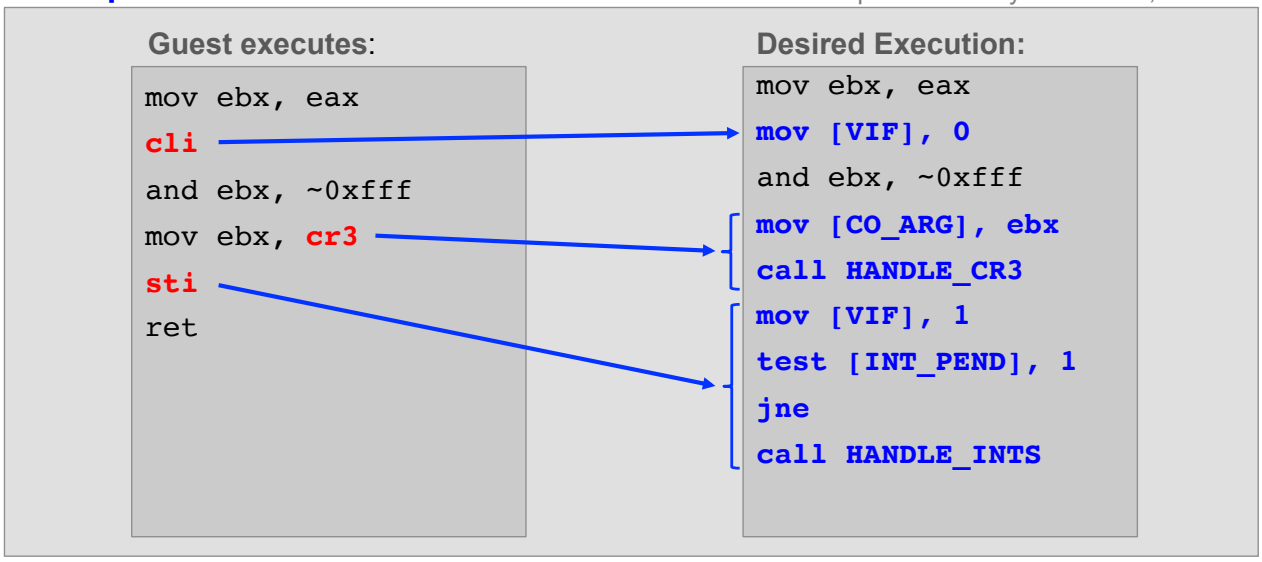
Virtualization - Mechanisms

- How does Virtualization work?
- Virtualization: What is the issue?
- Virtualization Techniques:
  - De-privileging ("Trap & Emulate")
  - Para-Virtualization
  - Binary Translation

Virtualization – What is the Issue?

Example:

Example from Vitaly Shmatikov, Cornell



Techniques in Classical Virtualization

1. **De-privileging** ("trap-and-emulate")
   - All instructions that read/write privileged state trap when executed in unprivileged level.
   - Execute guest OS directly, but at unprivileged level.
2. **Para-Virtualization**
   - "Modify quest operating system to provide higher-level information to VMM."
3. **Interpretive Execution**
   - Add dedicated HW execution mode for running the guest OS.
   - e.g. IBM 370 SIE ("start interpretive execution") instruction.
   - Reduces number of required traps.
4. **Binary Translation**
   - Used in WMWare.

Techniques in Classical Virtualization

1. **De-privileging** ("Trap-and-Emulate")
   - All instructions that read/write privileged state trap when executed in unprivileged level.
   - Execute guest OS directly, but at unprivileged level.
2. **Para-Virtualization**
   - "Modify quest operating system to provide higher-level information to VMM."
3. **Interpretive Execution**
   - Add dedicated HW execution mode for running the guest OS.
   - e.g. IBM 370 SIE ("start interpretive execution") instruction.
   - Reduces number of required traps.
4. **Binary Translation**
   - Used in WMWare.

Trap&Emulate

Example:

ov, Cornell

Guest executes:

```
mov ebx, eax
cli
and ebx, ~0xfff
mov ebx, cr3
sti
ret
```

```
mov [VIF], 0
iret
```

```
mov [CO_ARG], ebx
call HANDLE_CR3
iret
```

```
mov [VIF], 1
test [INT_PEND], 1
jne
call HANDLE_INTS
iret
```

Trap&Emulate: Definitions

Privileged State: Part of system state that determines resource allocation, e.g., addressing context, processor mode.

Control Sensitive Instruction: Changes privileged state.
Example: manipulate status register,
return from interrupt

Behavior Sensitive Instruction: Exposes privileged state.
Example: load physical address

Sensitive Instruction: Control or behavior sensitive

Formal Requirements for Virtualizable Third Generation Architectures

Gerald J. Popek
University of California, Los Angeles
and
Robert P. Goldberg
Honeywell Information Systems and
Harvard University

Virtual machine systems have been implemented on a limited number of third generation computer systems, e.g. CP-67 on the IBM 360/67. From previous empirical studies, it is known that certain third generation computer systems, e.g. the DEC PDP-10, cannot support a virtual machine system. In this paper, model of a third-generation-like computer system is developed. Formal techniques are used to derive precise sufficient conditions to test whether such an architecture can support virtual machines.

Key Words and Phrases: operating system, third generation architecture, sensitive instruction, formal requirements, abstract model, proof, virtual machine, virtual memory, hypervisor, virtual machine monitor
CR Categories: 4.32, 4.35, 5.21, 5.22

This is a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15–17, 1973.

This research was supported in part by the U.S. Atomic Energy Commission, Contract No. AT(11-1) Gen 10, Project 14 and in part by the Electronic Systems Division, U.S. Air Force, Hanscom Field, Bedford, Massachusetts under Contract Number F19628-70-0217.

Authors' addresses: Gerald J. Popek, Computer Science Department, University of California, Los Angeles CA 90024; Robert P. Goldberg, Honeywell Information Systems, Waltham, MA 02154.

Communications of the ACM
July 1974
Volume 17
Number 7

Trap&Emulate: Definitions

Privileged State: Part of system state that determines resource allocation, e.g., addressing context, processor mode.

Control Sensitive Instruction: Changes privileged state.

Behavior Sensitive Instruction: Exposes privileged state.

"Innocuous" Instruction: not sensitive.

Privileged Instruction: Traps when executed in user rather than in kernel mode (NOOP not sufficient!)

Formal Requirements for Virtualizable Third Generation Architectures

Gerald J. Popek
University of California, Los Angeles
and
Robert P. Goldberg
Honeywell Information Systems and
Harvard University

Virtual machine systems have been implemented on a limited number of third generation computer systems, e.g. CP-67 on the IBM 360/67. From previous empirical studies, it is known that certain third generation computer systems, e.g. the DEC PDP-10, cannot support a virtual machine system. In this paper, model of a third-generation-like computer system is developed. Formal techniques are used to derive precise sufficient conditions to test whether such an architecture can support virtual machines.

Key Words and Phrases: operating system, third generation architecture, sensitive instruction, formal requirements, abstract model, proof, virtual machine, virtual memory, hypervisor, virtual machine monitor
CR Categories: 4.32, 4.35, 5.21, 5.22

This is a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15–17, 1973.

This research was supported in part by the U.S. Atomic Energy Commission, Contract No. AT(11-1) Gen 10, Project 14 and in part by the Electronic Systems Division, U.S. Air Force, Hanscom Field, Bedford, Massachusetts under Contract Number F19628-70-0217.

Authors' addresses: Gerald J. Popek, Computer Science Department, University of California, Los Angeles CA 90024; Robert P. Goldberg, Honeywell Information Systems, Waltham, MA 02154.

Communications of the ACM
July 1974
Volume 17
Number 7

Trap&Emulate: Requirements

Theorem:
"For any conventional third generation [1974] computer, a virtual machine monitor may be constructed if the set of **sensitive** instructions for that computer is a **subset** of the set of **privileged** instructions."

In English:
A system is virtualizable by Trap & Emulate if all sensitive instructions are privileged.

Note: An instruction is sensitive when it is control or behavior sensitive.

Formal Requirements for Virtualizable Third Generation Architectures

Gerald J. Popek
University of California, Los Angeles
and
Robert P. Goldberg
Honeywell Information Systems and
Harvard University

Virtual machine systems have been implemented on a limited number of third generation computer systems, e.g. CP-67 on the IBM 360/67. From previous empirical studies, it is known that certain third generation computer systems, e.g. the DEC PDP-10, cannot support a virtual machine system. In this paper, model of a third-generation-like computer system is developed. Formal techniques are used to derive precise sufficient conditions to test whether such an architecture can support virtual machines.

Key Words and Phrases: operating system, third generation architecture, sensitive instruction, formal requirements, abstract model, proof, virtual machine, virtual memory, hypervisor, virtual machine monitor
CR Categories: 4.32, 4.35, 5.21, 5.22

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This is a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15–17, 1973.

This research was supported in part by the U.S. Atomic Energy Commission, Contract No. AT(11-1) Gen 10, Project 14 and in part by the Electronic Systems Division, U.S. Air Force, Hanscom Field, Bedford, Massachusetts under Contract Number F19628-70-0217.

Authors' addresses: Gerald J. Popek, Computer Science Department, University of California, Los Angeles CA 90024; Robert P. Goldberg, Honeywell Information Systems, Waltham, MA 02154.

Communications of the ACM
July 1974
Volume 17
Number 7

Two Obstacles to T&E Virtualization

Examples are for x86.

Visibility of Privileged State

- Current Privilege Level is stored in code segment register. Guest therefore can know that it runs in de-privileged mode.
- Interrupt descriptor table in virtual memory.

Lack of Traps when Privileged Instructions run at User-Level

- Some control-sensitive instructions generate NOOP in user mode rather than generating a trap.
- e.g. `POPF` "pop flags", which modifies ALU and system flags, must generate trap for VMM to intervene.

Example – More Troubles with `POPF`

Lack of Traps when Privileged Instructions run at User-Level

- Some control-sensitive instructions generate NOOP in user mode rather than generating a trap.
- e.g. `POPF` "pop flags", which modifies ALU and system flags, must generate trap for VMM to intervene.

x86 allows POPF in user mode, but simply does not update interrupt flag (IF) in the system flags if executed in user mode. (This protects OS from mis-behaving users programs.)

If operating system runs in user mode, it has no way to know whether to enable interrupts or not.

Techniques in Classical Virtualization

1. **De-privileging** ("trap-and-emulate")
   - All instructions that read/write privileged state trap when executed in unprivileged level.
   - Execute guest OS directly, but at unprivileged level.
2. **Para-Virtualization**
   - "Modify quest operating system to provide higher-level information to VMM."
3. **Interpretive Execution**
   - Add dedicated HW execution mode for running the guest OS.
   - e.g. IBM 370 SIE ("start interpretive execution") instruction.
   - Reduces number of required traps.
4. **Binary Translation**
   - Used in WMWare.

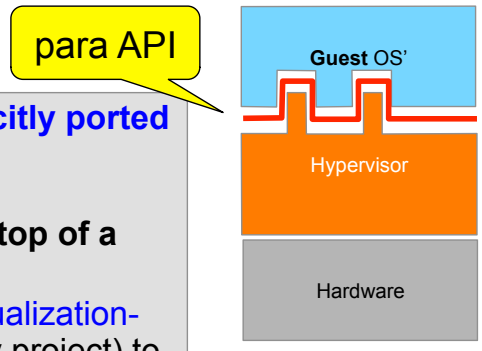
Virtualization Techniques: Paravirtualization

Present software interface (the "para-API") to virtual machines that is similar but not identical to that of the underlying hardware.

Provide specially defined 'hooks' to allow the guest(s) to hand over handling of problematic portions of code to VMM.

Requires the guest operating system to be **explicitly ported** for the **para-API.**

- **A conventional O/S distribution that is not paravirtualization-aware cannot be run on top of a paravirtualized VMM**!
- Xen solution for closed-source O/Ss: paravirtualization-aware device drivers (e.g. XenWindowsGplPv project) to be installed in guest O/S.



Techniques in Classical Virtualization

1. **De-privileging** ("trap-and-emulate")
   - All instructions that read/write privileged state trap when executed in unprivileged level.
   - Execute guest OS directly, but at unprivileged level.
2. **Para-Virtualization**
   - "Modify quest operating system to provide higher-level information to VMM."
3. **Interpretive Execution**
   - Add dedicated HW execution mode for running the guest OS.
   - e.g. IBM 370 SIE ("start interpretive execution") instruction.
   - Reduces number of required traps.
4. **Binary Translation**
   - Used in WMWare.

VMware Software VMM: Binary Translation

Observation: Traditionally, software VMMs run very slow due to interpretation.

Binary Translation:

- Replace sensitive instructions in guest binary on-the-fly and replace by emulation code.
- Binaries as input, not source code.
- Dynamic translation at run-time.
- Input is full x86 instruction set. Output is safe subset.

Binary Translation: Simple Example

```
int isPrime(int a) {
  for (int i = 2; i < a; i++) {
    if (a % i == 0) return 0;
  }
  return 1;
}
```

<- small example, C code

Binary Translation: Simple Example

```
int isPrime(int a) {
  for (int i = 2; i < a; i++) {
    if (a % i == 0) return 0;
  }
  return 1;
}
```

<- small example, C code

same code, compiled ->

```
isPrime:    mov   %exc, %edi   ; %ecx = %edi (argument a)
            mov   %esi, $2     ; i = 2
            cmp   %esi, %ecx   ; is i >= a?
            jge   prime        ; jump if yes
nexti:      mov   %eax, %ecx   ; set %eax = a
            cdq                ; sign-extend
            idiv  %esi         ; a % i
            test  %edx, %edx   ; is remainder zero?
            jz    %notPrime    ; jump is yes
            inc   %esi         ; i++
            cmp   %esi, %ecx   ; is i >= a?
            jl    nexti        ; jump if no
prime:      mov   %eax, $1     ; return value in %eax
            ret
notPrime:   xor   %eax, %eax   ; %eax = 0
            ret
```

Binary Translation: Mechanics

Instruction stream

```
89 f9 be 02 00 00 00 39 ce 7d ...
```

Translation Unit (TU)

```
isPrime:    mov   %exc, %edi
            mov   %esi, $2
            cmp   %esi, %ecx
            jge   prime
```

1. read prefixes, opcodes, operands
2. stop at 12 instructions or terminating instruction (control flow) -> **Translation Unit** (TU)
3. leave innocuous instructions alone
4. translate remaining instructions
5. generate **compiled-code-fragment** (CCF)

```
isPrime':   mov   %exc, %edi    ; IDENT
            mov   %esi, $2
            cmp   %esi, %ecx
            jge   [takenAddr]   ; JCC
            jmp   [fallthrAddr]
```

Compiled-Code Fragment (CCF)

Translation Result

```
isPrime:    mov   %exc, %edi   ; %e
            mov   %esi, $2     ; i
            cmp   %esi, %ecx   ; is
            jge   prime        ; ju
nexti:      mov   %eax, %ecx   ; se
            cdq                ; si
            idiv  %esi         ; a
            test  %edx, %edx   ; is
            jz    notPrime     ; jum
            inc   %esi         ; i+
            cmp   %esi, %ecx   ; is
            jl    nexti        ; ju
prime:      mov   %eax, $1     ; re
            ret
notPrime:   xor   %eax, %eax   ; %e
            ret
```

```
isPrime':   mov   %exc, %edi   ; IDENT
            mov   %esi, $2
            cmp   %esi, %ecx
            jge   [takenAddr] ; JCC
                              ; fall-through into next CFF
nexti':     mov   %eax, %ecx   ; IDENT
            cdq
            idiv  %esi
            test  %edx, %edx
            jz    notPrime'    ; JCC
                              ; fall-through into next CFF
            inc   %esi         ; IDENT
            cmp   %esi, %ecx
            jl    nexti'       ; JCC
            jmp  [fallthrAddr3]

notPrime':  xor   %eax, %eax   ; IDENT
            pop   %r11         ; RET
            mov   %gs:0xff39eb8(%rip), %rcx ; spill $rcx
            movzx %ecx, $r11b
            jmp   %gs:0xfc7dde0(8*%rcx)
```

Binary Translation: Observations

Observation: This approach scales well:

- e.g., Windows XP boot/halt translates to
  - 229,347 64-bit translation units (TUs) of up to 12 instructions.
  - 23,909 32-bit TUs
  - 6,680 16-bit TUs

Observation: Translated code has good instruction-cache locality.

- Translator captures execution trace of guest code.
- Rarely-executed code (e.g. error handling) is placed off the "hot" execution path.

Most Instructions need no Translation, except

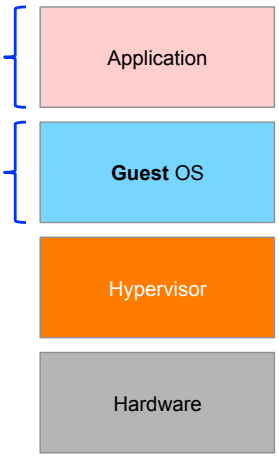
1. Instructions that are affected by translation, because code layout changes:
   - PC-relative addressing
   - Direct control flow (direct calls, branches, jumps)
   - Indirect control flow (`jmp`, `call`, `ret`)

2. Sensitive instructions:
   - Some instructions run faster in binary translation mode than native.
     - e.g. `cli` (clear interrupts) on Pentium 4 takes 60 cycles; replaced by `vcpu.flags.IF:=0`.
   - Other operations (e.g. context switch) may need to call out to a runtime, with lots of overhead.

What about User-Level Code?

Use "direct execution" of guest user-code

Use Binary Translation of guest kernel-code



Observation: Binary Translation is not needed for safe execution of "most user code" on "most guest operating systems".

Virtualization - Mechanisms

- How does Virtualization work?
- Virtualization: What is the issue?
- Virtualization Techniques:
  - De-privileging ("Trap & Emulate")
  - Para-Virtualization
  - Binary Translation