

# A Choices Hypervisor on the ARM architecture

Rishi Bhardwaj, Phillip Reames, Russell Greenspan  
Vijay Srinivas Nori, Ercan Ucan

## ABSTRACT

*Choices is an object oriented operating system that runs on the x86 and ARM architectures. The aim of this project is to build a hypervisor using Choices on the ARM architecture. In this project we aim to build a hypervisor on the QEMU emulator emulating the ARM7 architecture using a VMX approach. We have selected as guest operating system a small Linux image. The primary goal of the system is to emulate the sensitive instructions of the ARM architecture using VMX like instructions in the hypervisor. We have identified the sensitive instructions for the ARM architecture. We have also put in place a mechanism by which QEMU traps on sensitive instruction to the hypervisor. In future, we will proceed to implement the validation and emulation of the sensitive instructions in Choices while extending QEMU with the necessary VMX instructions.*

## 1. INTRODUCTION

A Virtual Machine Monitor (VMM) or hypervisor is an example of system software for a computer system that creates efficient, isolated programming environments that provide users with the appearance of direct access to the real machine environment. In today's world, the amount of computing resources available is substantially larger than what one OS would typically need. Therefore, an important application of the available computing power is to have more than one OS sharing the same hardware in a secure, reliable and scalable manner. VMMs have full control of the processor and provide an abstraction of a virtual processor to the guest software (operating system) and to supervise the execution of the guest. Thus the VMM provides a guest operating system the illusion of being the sole user of the machine. The VMM manages the various physical hardware resources of the machine which are shared between the different guest Operating Systems.

## 2. BACKGROUND

In order to have our midterm position report make sense to the reader, we thought that it would be useful to provide some background information about the system that we are working on. As mentioned in the preceding sections of the report, in the scope of this research, we will be taking a VMX type of virtualization approach on ARM architecture. In some sense, we will be changing the ARM hardware. Because there is no virtualizable hardware that we are aware of for ARM, instead of dealing with hardware, we will be working with an application called QEMU that behaves like the hardware.

### 2.1 Need for Virtualization

Operating System (OS) virtualization [7] allows multiple OSes to run in secure, isolated environments on shared hardware. Since a Virtual Machine Monitor (VMM) or Hypervisor controls access to shared resources, such as peripheral devices and memory, we can be assured that each OS can only access its own data. This allows hardware resources to be optimally multiplexed for overall OS workload, leading to server consolidation and reduced maintenance costs. Additionally, this affords us an ideal way to run mistrusted applications and unverified code in secluded environments.

### 2.2 VMX Approach

Virtual Machine Extensions (VMX) for IA-32 processors define processor-level support for virtual machines on IA-32 processors. The virtual machine architecture supports two principle classes of software:

1. Virtual Machine Monitor (VMM)
2. Guest Software/OS.

The VMX adds a new processor operation called VMX operation. There are two kinds of VMX operations: root and non root. The VMM runs in the VMX root operation while the guest OS runs in the VMX non root operation. The behavior of the processor in the VMX root operation is very much the same as outside the VMX operation. The difference being the addition of certain new instructions (VMX instructions) and limited access to certain control registers. The processor behavior in non root VMX operation can be restricted or modified to facilitate virtualization. Depending upon the configuration of the processor set by the VMM, certain instructions, when executed in the non root VMX operation, would cause trap into VMM. This helps the VMM

to regain control of the hardware and emulate the sensitive instruction requested by the guest software in the non root VMX operation. The goal of the whole process being that the guest software should not be able to determine that it is running in a virtual machine. The above goal is defeated in the current IA-32 processors where certain non privileged instructions give away the privilege level the processor is currently executing in, for more details please see [1].

VMX defines a new control structure called Virtual Machine Control Structure (VMCS) to manage the transitions into and out of non-root operation (VM entries and VM exits) as well as processor operation in VMX non root operation. Special instructions are added to modify this control structure.

### 2.3 ARM7 Architecture

The ARM architecture [2] is one of the most popular architectures for handheld and embedded devices. ARM stands for Advanced RISC Machines Ltd. ARM is a 32-bit RISC architecture and is widely used in applications where saving power is critical. One of the advantages of writing a hypervisor for ARM is that we have a small instruction set and the number of sensitive instructions is very small, leading to a smaller, easier to verify and easy to develop hypervisor. The ARM architecture has six basic operating modes:

1. User: unprivileged mode under which most tasks run or the normal program execution state.
2. FIQ: entered when a high priority (fast) interrupt is raised for data transfer or channel purposes.
3. IRQ: used for general purpose interrupt handling
4. Supervisor: entered on reset and when a Software Interrupt instruction is executed.
5. Abort: used to handle memory access violations or on instruction prefetch abort.
6. Undef: used to handle undefined instructions

ARM has 37 registers all of which are 32-bits long.

1. 1 dedicated program counter
2. 1 dedicated current program status register
3. 5 dedicated saved program status registers
4. 30 general purpose registers

The current processor mode governs which of several banks is accessible. Each mode can access

1. a particular set of r0-r12 registers
2. a particular r13 (the stack pointer, sp) and r14 (the link register, LR)

3. the program counter, r15 (PC)
4. the current program status register, CPSR

Privileged modes (except System) can also access a particular SPSR (saved program status register)

When an exception occurs, the ARM processor does the following tasks:

1. Copies CPSR into corresponding SPSR\_<mode>
2. Sets appropriate CPSR bits
3. Change to ARM state
4. Change to exception mode
5. Disable interrupts (if appropriate)
6. Stores the return address in LR\_<mode>
7. Sets PC to vector address

To return, exception handler needs to:

1. Restore CPSR from SPSR\_<mode>
2. Restore PC from LR\_<mode>
3. This can only be done in ARM state.

### 2.4 Intro to QEMU

QEMU is an open source processor emulator. Basically, it uses dynamic translation to achieve good emulation speed. Unlike VMWARE and Win4Lin, it emulates the CPU instead of only virtualizing the computer. This means that it is considerably slower, but on the other hand it is much more portable, stable and secure [3].

It has two operating modes [4]:

- Full system emulation. In this mode, QEMU emulates a full system (for example a PC), including one or several processors and various peripherals. It can be used to launch different Operating Systems without rebooting the PC or to debug system code.
- User mode emulation (Linux host only). In this mode, QEMU can launch Linux processes compiled for one CPU on another CPU. It can be used to launch the Wine Windows API emulator [6] for to ease cross-compilation and cross debugging.

Moreover, it is an emulator that can run without any host kernel driver and yet gives acceptable performance. In system emulation QEMU supports ARM Integrator/CP (Arm 1026E Processor) that we are targeting to virtualize. It supports ARM architecture for user emulation mode as well. We are interested in using this mode while checking our test cases. (Detailed info about test cases will be presented in proceeding sections of this report.) QEMU is quite generic in the sense that it has [5]:

- User space only or full system emulation (as stated above).
- Self-modifying code support.
- Precise exceptions support.
- The virtual CPU, which is a library (libqemu) that can be used in other projects.

As mentioned above, QEMU is an open source software and this makes it a very useful tool for operating system research. In our project, we are working on the ARM Integrator/CP emulator package and working both with the full system emulation mode and the user mode emulation. For the whole of the project we work with the former mode of emulation since we want the hardware emulation in the sense that we want to modify this ‘virtual’ hardware. For the small test cases (i.e. small assembly programs to test our modifications on QEMU) that we write, we will be utilizing the user mode emulation as well. This will help us do the testing in some sense in a “modular” way.

### 3. CONTRIBUTIONS

#### 3.1 ARM sensitive Instructions

An operating system kernel runs in a mode with higher privileges (usually the highest privileges) compared to the user applications. This higher privileged mode allows the OS access to certain machine instructions (called privileged instructions) and resources not available to the user mode applications. This creates a problem in building a hypervisor for the architecture. The OS will now be run in a mode by the hardware in a semi-privileged mode, with traps on sensitive instructions going to the hypervisor. It is the responsibility of the hypervisor to keep track of the OS’s state, allow the safe operations to execute and block the illegal sensitive operations from executing. Attempts at analyzing the problem of how to virtualize the Intel Pentium architecture [1] show that the instructions that any architecture should consider as sensitive satisfy the following conditions:

1. An instruction represented by the same bit pattern or format executes differently in modes of different privileges. This means that on execution, the instruction must trap to the hypervisor which has the mode information to execute the instruction properly.
2. A instruction who’s execution requires greater privileges than the privilege in which the guest OS is being run.

Using this as a basis we classify the following instructions in the ARM7 architecture [2] as sensitive and present our reasons for the classification. It should be noted that these instructions that are being recognized as sensitive have several variants based on the instruction condition codes (each having the same opcode) e.g. LDM and LDMFD (a special condition code set). We’re just listing the main instructions here and not all the variants

1. These instructions touch the CPSR (one cardinal rule in sensitive instruction identification in hypervisor design is that the guest OS should not know that it is being run as guest and NOT on the actual hardware itself). Consequently, the hypervisor must validate them and adjust the OS’s state accordingly.
  - (a) MRS: Moves PSR status/flags to register.
  - (b) MSR: Moves register to PSR status/flags.
2. An instruction in the same format has different effects on execution in user and supervisor modes. The hypervisor must track which mode the guest OS was to be in to execute the instruction in the appropriate mode.
  - (a) TEQ: (in TEQP form) Test bitwise equality
  - (b) STM: Stores Multiple registers to stack (push)
  - (c) LDM: Loads multiple registers from stack (pop)
3. This instruction is used to switch mode from ARM to Thumb (We are not supporting Thumb for now)
  - (a) BX: Branch with exchange, Switches processor mode to Thumb
4. This instruction is used to switch mode from user to supervisor.
  - (a) SWI: Software Interrupt
5. These instructions are coprocessor instructions and directly touch the MMU and page tables. Hence, they are sensitive.
  - (a) LDC: Load coprocessor register from memory
  - (b) STC: Store coprocessor register to memory
  - (c) MRC: Move coprocessor register to CPU register
  - (d) MCR: Move CPU register to coprocessor register
  - (e) CDP: Coprocessor-specific data processing instruction
6. These instructions are not sensitive unless they attempt to branch into a region of memory not controlled by the guest OS but this will not be possible due to the virtual memory address translation. Each OS will have a complete address space that will be translated by the virtual memory system into unique physical addresses.
  - (a) BL: Branch with Link
  - (b) B: Branch

#### 3.2 Changes to QEMU

For emulating ARM machine on an x86 host system, QEMU translates each ARM instruction into a set of corresponding x86 instructions. The translation, done at runtime, is batched with a ‘block’ of ARM instructions translated together in one go. Instructions such as a branch or SWI (or anything that might change the flow of execution) mark the end of a ARM instruction block. The translation process is stopped upon encountering a block boundary after which the translated x86 code is executed on the host machine. After execution the translated “block” is cached so that the dynamic translation does not have to be done every time the

same ARM instructions are to be run. The whole process is then repeated for the next block of ARM code.

Our mid term aim was to provide a mechanism in QEMU for control transfer from a guest OS to the hypervisor if and when a sensitive instruction needed to be executed. Modifications in the QEMU code to achieve the above desired effect have been done. After making the changes, QEMU now treats(translate) the sensitive instructions, when running in less privileged guest OS mode, like a SWI (software interrupt) with slight modifications. With the help of these modifications the control can now be transferred to a pre-defined Hypervisor's hyper call routine whenever emulation of a sensitive instruction is required. If the CPU is running in hypervisor-privilege mode, then all instructions are executed unaltered.

### 3.3 Tests & Results

We tested our modifications in QEMU by running user programs with privileged and unprivileged ARM sensitive instructions in them. The ARM code was appropriately translated to x86 code. While executing the translated x86 code the sensitive instructions rightly give a trap, upon which a message is displayed on screen specifying which sensitive instruction was tried to be executed. All the user programs were run with user mode emulation of qemu. Also executing kernel code (linux image run on qemu full arm system emulation) does not give any such output on screen.

## 4. RELATED WORK

Our approach, modifying QEMU to trap on specific instructions, is a simulation of processor-level support for virtualization. To our knowledge, no such hardware-based virtualization has been attempted on the ARM processor. However, two recent hypervisors from French companies, TRANGO and Jaluna, offer software-based virtualization on the ARM processor, and the ARM Corporation has added hardware-based primitives for process isolation via TrustZone, which could have future impacts on OS virtualization.

TRANGO [8] is a 20KB ROM-able microkernel that allows multiple guest OSes to run on a single ARM9 processor. The hypervisor divides resources and address space and runs guest OSes in "virtual kernel mode", a logical division of CPU-user mode. As such, only the TRANGO footprint runs in CPU-kernel mode.

Jaluna OSware [9] provides virtualization on ARM9 processors optimized for cross-OS communication. The Jaluna Engine, targeted for mobile devices simultaneously running a real-time OS and a Linux OS, does not isolate virtual machines but instead provides virtualization of hardware resources while keeping each OS running in kernel-mode, reducing security but improving cross-OS communication.

Though not truly a virtualization scheme, ARM's own TrustZone [10] promises hardware-based support for running code in secure, isolated environments. TrustZone provides protection for interrupt handlers, peripheral device access, and off-chip memory by maintaining hardware separation between secure and non-secure information. This ensures that rogue processes can never access intra-process sensitive data.

## 5. FUTURE WORK

A preview of the design and implementation in the next phase of the project:

- Add the necessary functionality to the Choices hypervisor in order to allow the hypervisor to perform necessary validations on the sensitive instructions for which the trapping mechanism from QEMU is already in place.
- Add extra instructions to manage virtual machines and enable and disable the hypervisor mode.
- Formulate the experiment set in the form of a comprehensive test suite for the system to measure its performance on standard benchmarks (and for validation).

## 6. REFERENCES

- [1] J.Robin, C.Irvine, "Analysis of the Intel Pentium's ability to Support a Secure Virtual Machine Monitor", *In Proceedings of the 9th USENIX Security Symposium, Denver, CO, USA, pages 129-144, Aug 2000.*
- [2] ARM7 TDMI data sheet, <http://www.e-lab.de/ARM7/ARM-instructionset.pdf>
- [3] QEMU information on Wiki, [http://wiki.archlinux.org/index.php/Qemu#Choosing\\_Qemu\\_version](http://wiki.archlinux.org/index.php/Qemu#Choosing_Qemu_version)
- [4] QEMU user documentation, <http://fabrice.bellard.free.fr/qemu/qemudoc.html#SEC1>
- [5] QEMU technical documentation, <http://fabrice.bellard.free.fr/qemu/qemutech.html#SEC1>
- [6] Wine Windows API emulator, <http://www.winehq.org>
- [7] An Introduction to Virtualization, <http://www.kernelthread.com/publications/virtualization>
- [8] Trango System Website, <http://www.trango-systems.com>.
- [9] Jaluna - Virtual Infrastructure Software for Connected Devices, <http://www.jaluna.com>.
- [10] TrustZone Technology Overview, <http://www.arm.com/trustzone>.

## 7. APPENDIX

### ARM7 Instruction Set

Mnemonic	Instruction	Action
ADC	Add with carry	$Rd := Rn + Op2 + Carry$
ADD	Add	$Rd := Rn + Op2$
AND	AND	$Rd := Rn \text{ AND } Op2$
B	Branch	$R15 := \text{address}$
BIC	Bit Clear	$Rd := Rn \text{ AND NOT } Op2$
BL	Branch with Link	$R14 := R15, R15 := \text{address}$
BX	Branch and Exchange	$R15 := Rn,$ T bit := $Rn[0]$
CDP	Coprocessor Data Processing	(Coprocessor-specific)
CMN	Compare Negative	$CPSR \text{ flags} := Rn + Op2$
CMP	Compare	$CPSR \text{ flags} := Rn - Op2$
EOR	Exclusive OR	$Rd := (Rn \text{ AND NOT } Op2)$ $OR(Op2 \text{ AND NOT } Rn)$
LDC	Load coprocessor from memory	Coprocessor load
LDM	Load multiple registers	Stack manipulation (Pop)
LDR	Load register from memory	$Rd := (\text{address})$
MCR	Move CPU register to coprocessor register	$cRn := rRn <op> cRm$
MLA	Multiply Accumulate	$Rd := (Rm * Rs) + Rn$
MOV	Move register or constant	$Rd := Op2$
MRC	Move from coprocessor register to CPU register	$Rn := cRn <op> cRm$
MRS	Move PSR status/flags to register	$Rn := PSR$
MSR	Move register to PSR status/flags	$PSR := Rm$
MUL	Multiply	$Rd := Rm * Rs$
MVN	Move negative register	$Rd := 0xFFFFFFFF \text{ EOR } Op2$
ORR	OR	$Rd := Rn \text{ OR } Op2$
RSB	Reverse Subtract	$Rd := Op2 - Rn$
RSC	Reverse Subtract with Carry	$Rd := Op2 - Rn - 1 + Carry$
SBC	Subtract with Carry	$Rd := Rn - Op2 - 1 + Carry$
STC	Store coprocessor register to memory	$\text{address} := CRn$
STM	Store Multiple	Stack manipulation (Push)
STR	Store register to memory	$<\text{address}> := Rd$
SUB	Subtract	$Rd := Rn - Op2$
SWI	Software Interrupt	OS call
SWP	Swap register with memory	$Rd := [Rn], [Rn] := Rm$
TEQ	Test bitwise equality	$CPSR \text{ flags} := Rn \text{ EOR } Op2$
TST	Test bits	$CPSR \text{ flags} := Rn \text{ AND } Op2$