VERTIGO

Controlling a Smartphone from an Architectural Vantage Point

Kirk Swidowski

mail *kirk@swidowski.com*
web *www.de7ec7ed.com*
twitter *de7ec7ed*
github *github.com/de7ec7ed*

Special thanks to Dr. Joseph Sharkey for his help with the project

About me

- Principal Scientist at Siege Technologies
- Expertise includes:
  - Virtualization (x86 [Intel-VT/AMD SVM] and ARM)
  - Trusted Computing (Intel TXT, ARM TrustZone, TPM)
  - Boot Technologies
  - Computer Architecture
  - Reverse Engineering
- Prior work includes research & development on topics including:
  - Anti-Exploitation Protection Systems
  - Preventing Sensitive Information and Malicious Traffic from Leaving Computers
  - Dynamic *Rooted* Trust
- M.S. Computer Science
  - Focused on the design and construction of a pedagogical RISC based computer architecture, including the ISA, 4 stage scalar pipeline with branch prediction and precise exception/interrupt handling (presented at CCSCNE 2009)
- Mobile software was published in "*A Windows Mobile Wish List*", Smartphone & PocketPC Magazine

Agenda

- The Point Up Front
- Usage
- Background
- Microvisor Overview
- Kernel's Key Enabling Components
- Loaders Exposed
- Modules & Management
- Debugging Facilities
- Construction & Layout
- Questions

Lots of content, the hope is that they will be used as reference material after the presentation

The Point Up Front

- To provide an extendable microvisor, for the ARM architecture, that can be used for a variety of activities
- Expose the tool and techniques so they can't be used for malicious things
- Show that software can be constructed and compatible with an architecture instead of an OS
- **Recently Open Sourced and Released under GPL so others have a solid foundation to learn from and build upon**
- Others have discussed thin virtualization for x86 *(e.g. SubVirt, Vitrol, BluePill, VirtDbg) and ARM (e.g. Cloaker)*
  - Source code is unavailable for BluePill (available for BlackHat training only), Cloaker and Vitrol
  - Messy code that is not meant to be extended (magic numbers, not separated from the underlying OS correctly, no modularity, etc.)

Represents a different paradigm of compatibility

(Architecture instead of an Operating System)

Usage

Reverse Engineering

- Perform dynamic analysis (underlying OS user and kernel space)
  - Dump register and memory state a key points in time
  - Trap on memory read/write (watchpoints)
  - Trap on execution (breakpoints)
  - Exception/Interrupt Injection
- Debug kernel panics without the need of expensive hardware *(e.g. JTAG)* or built-in debuggers, such as KDB
- Discover undocumented hardware
  - Create drivers to manipulate them
- Dump and manipulate the page tables of an underlying OS
  - Show pages matching a particular set of attributes *(e.g. cache, buffered, user, kernel, execute never, etc.)*

Enhanced Security

- Protect an applications memory when it is not executing
- Inject an arbitrary application into the underlying OS
  - Exposed as a virtual file under the *"/proc"* file system (not located on non-volatile storage at runtime)
  - Executed by hooking the *schedule()* function in the kernel
- Provide *"out of band"* functionality, such as integrity checks in a generic OS agnostic way

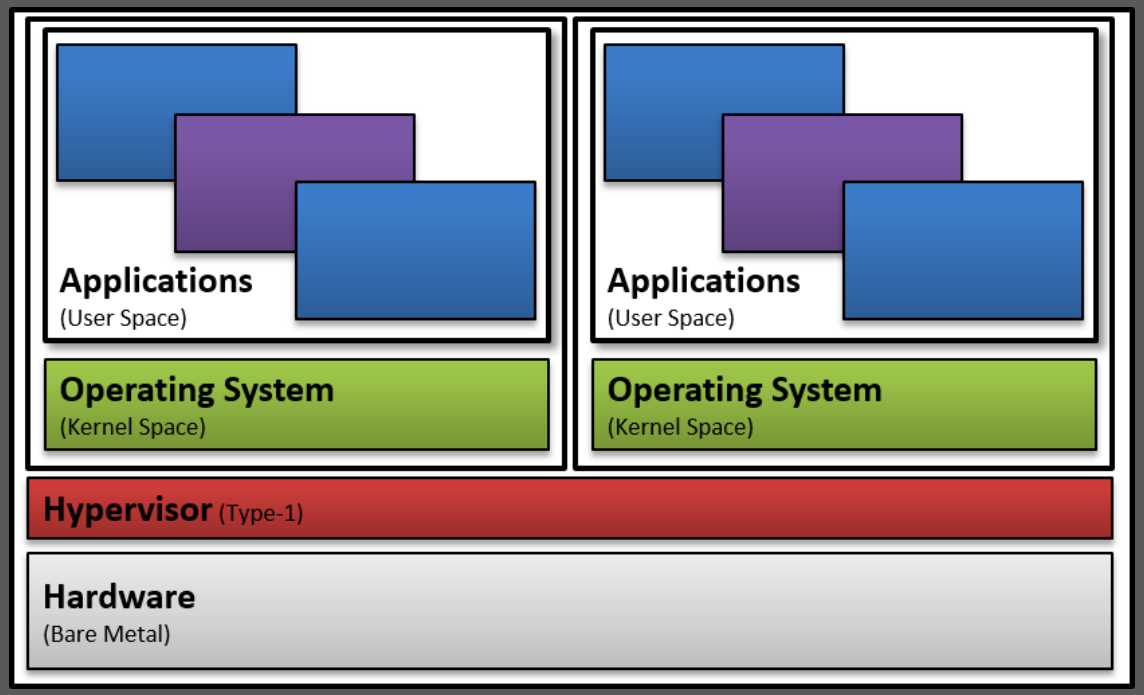
Advanced research in reverse engineering and security requires advanced tools, the hope is for the microvisor to provide an extendable foundation

Background

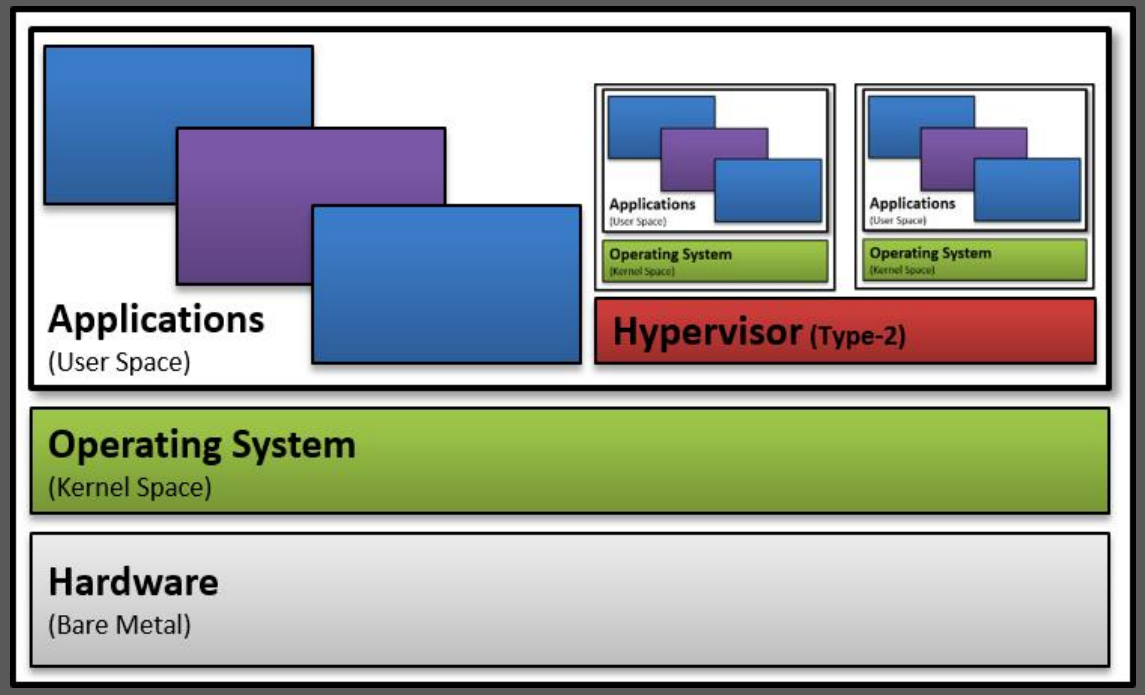
Describes the different types of hyper- /micro- visors and their position in a system's software stack.

Type-1 Hypervisors

- Resides under all operating systems, on the hardware *(e.g. native or bare metal)*
- Existing Products:
  - VMware ESXi
  - Xen
  - Microsoft Hyper-V
  - KVM (kind of)
- Can leverage hardware facilities *(e.g. ARM VE, Intel VT or AMD SVM)*



Type-2 Hypervisors

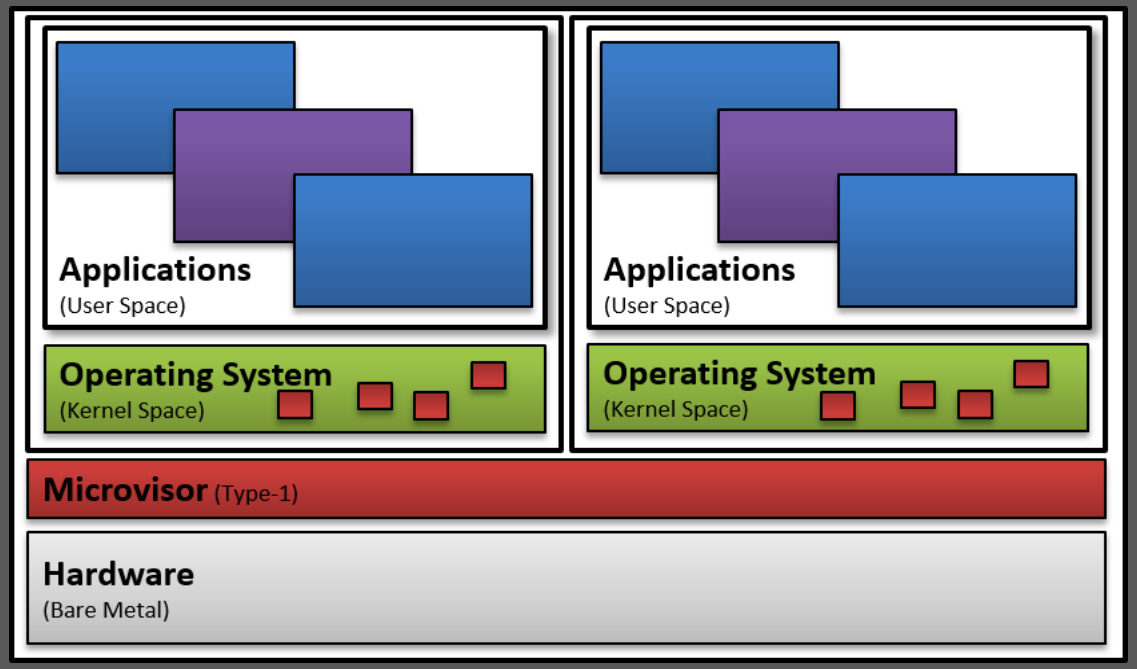


- Resides above the host operating system
- Existing Products:
  - VMware Workstation
  - VMware Fusion
  - Parallels Desktop
  - Oracle VirtualBox
  - QEMU
  - Bochs
  - Microsoft Virtual PC
- Can leverage hardware facilities *(e.g. ARM VE, Intel VT or AMD SVM)*

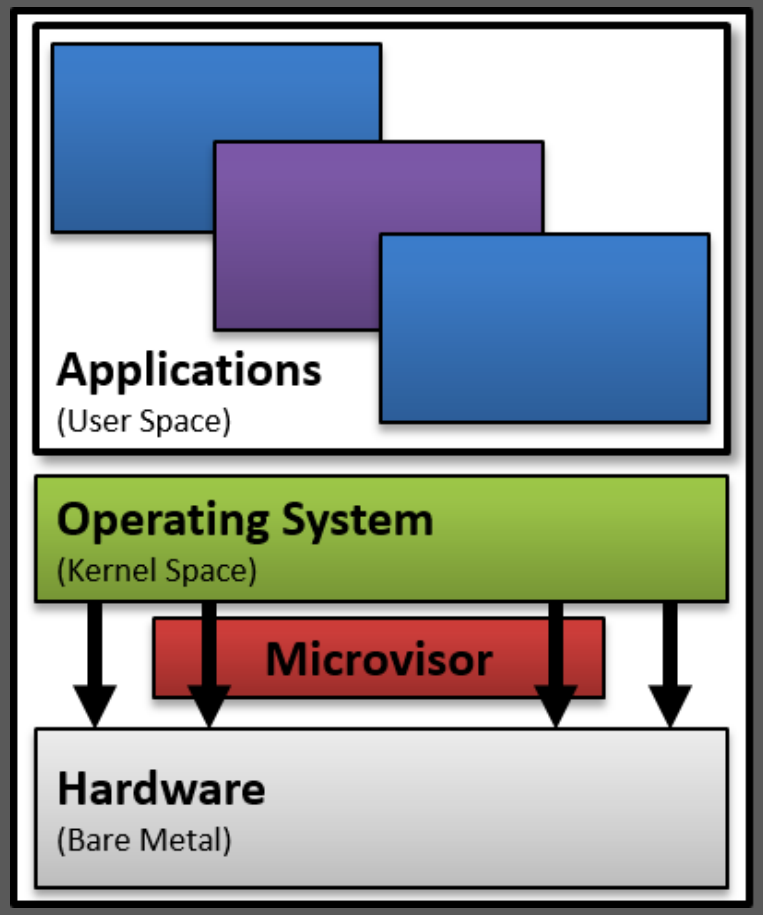
Microvisors

- Existing Products:
  - Open Kernel Labs (OKL)4
  - CODEZERO
- Paravirtualization
  - Requires source code modifications *(e.g. hypercalls)*
  - Adding the hypercalls to an OS is similar to adding support for a new architecture
- Twists architectural features to provide an additional layer of separation
  - Domain separation on ARM, de-privilege the VM OS (runs completely in user space)
    - VM user space and kernel space are separated into different domains

x86 hypervisors used segmentation to provide an additional layer of separation



Thin Virtualization (VERTIGO's Usage)



- Supports only a single OS, not intended to completely virtualize a system
- Leverage features and techniques of virtualization to accomplish tasks
- Does not require source code modifications of the underlying OS
- Uses para-passthrough techniques to maintain control without adversely affecting performance

Microvisor Overview

An Overview of the microvisor and its components.

Overview

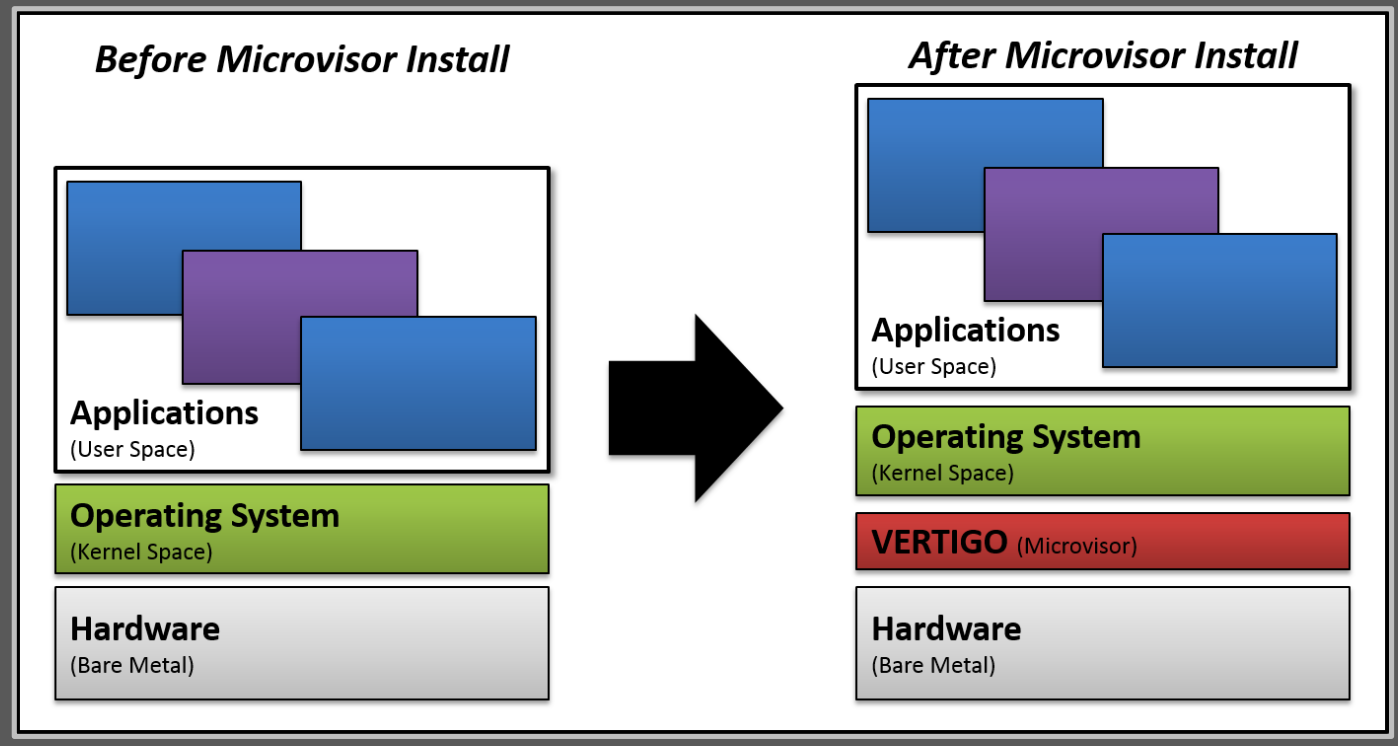
- **VERTIGO** – **V**irtualized **E**xtendable **R**untime **T**ransport *for* **I**ntegrated **G**ermane **O**perations
- Dynamically lifts a running commodity Operating System (OS) into a state analogous to a virtual machine (VM)
- Leverages *only* architectural features in the kernel to maintain broad compatibility with Cortex-A series processors and their supported OSs
- Constructed as a position independent flat binary to facilitate novel loading methods, such as kernel exploits

Features

- Kernel features include a:
  - Dynamic Linker
  - Vector Dispatcher
  - ¼ Size Paging Subsystem
  - Memory Allocation Subsystem
  - Debugging Subsystem
  - Numerous API to manipulate ARM architecture primitives
- **Supports runtime linking and module loading via a hidden communications channel**
- The kernel is kept generalized and modules are intended to load use-case specific code
  - Import API from the kernel and other modules
  - Export API for other modules

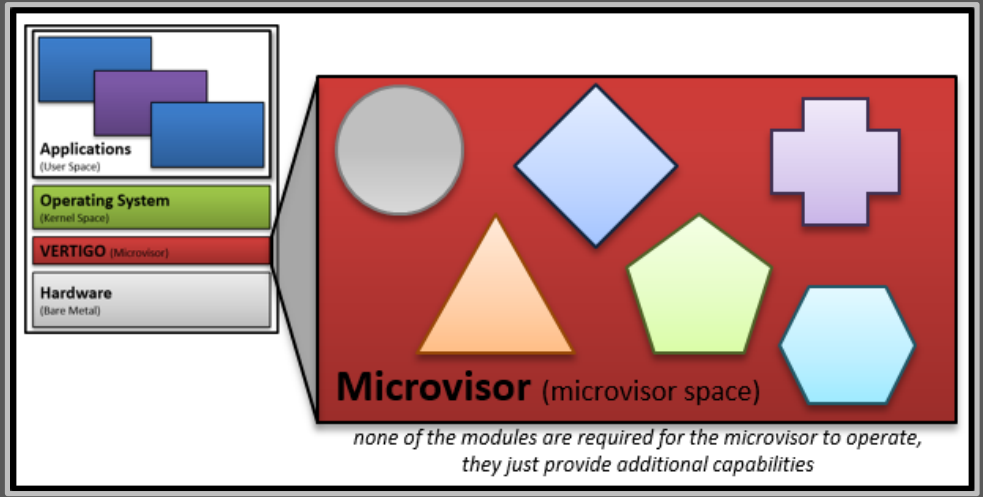
Installation

- Microvisor Requirements:
  - ARM Cortex A series SoC (tested on the A8 and A9)
  - A chunk of *virtually* contagious non-pageable memory
  - Execution in supervisor mode
- Supports a single commodity OS at a time
- Does *not* use any API in the underlying OS
- ***Installs itself on a live system by dynamically lifting the running OS in to a state analogous to a virtual machine***



Minimal requirements allow the microvisor to maintain clean separation from its underlying OS, facilitating portability between devices

Modularity



- Modules will be used to allow the microvisor to expand its footprint on the system once details have been identified *(e.g. leverage peripherals, OS API)*

- **Modularity is key to enable code reusability**
- SoCs and OSs are commonly interchanged
  - SoC X is used with OS Y and OS Z : OS X is used with SoC Y and SoC Z

Communication Module(s)
- Leverages Peripheral modules and/or OS Introspection to expose high level API for communication

Operating System Introspection Module(s)
- Exposes API for: Google Android, Apple iOS, Microsoft Windows RT, etc.

System On Chip (SoC) Module(s)
- Adds functionality specific to an SoC *(e.g. MSM8960, S5L8930, GIC PL192, GIC PL390)*

Peripheral Module(s)
- Provides basic abstraction to attached peripherals *(e.g. display, networking, GPS, USB)*

Light Weight Shadow Paging Module
- Provides the ability to intercept memory access *(e.g. Memory Mapped Input Output [MMIO])*

Protection Module(s)
- Protects arbitrary memory pages (*e.g. microvisor, kernel and/or application*)

Support

- Tested with the Apple iPhone 4 *(iOS 5.0.1 and 5.1.1)* and the Samsung Galaxy SIII *(Android 4.0.4)*
- It is installed at run-time through an OS specific loader
  - On iOS it is a user space application that will inject the microvisor into kernel space
  - On Android it is a kernel module
  - Any privilege escalation technique could be tailored to accommodate microvisor loading

Kernel's Key Enabling Components

Main points of the kernel. Highlights some of the more interesting parts of the microvisor.

Flat Binary

- A flat binary is a very plain file format in the context of this project it contains code and data
- **Constructing the microvisor as a flat binary provides separation the underlying OS and encourages portability**
- Common examples of this are shell code and boot loaders
- It does not retain any structures or sections from the build process *(e.g. import tables, file format headers [elf, pe, etc.])*
  - All sections are collapsed and placed back to back during the linking stage *(e.g. text, bss, data, rodata, LC0)*
  - Not all sections are included in the file as some are unnecessary
- The runtime base address of the binary can be specified in a linker script, if known
- If the runtime base address is unknown *(most cases, right ☺)* global address in the binary file will be incorrect
  - Specifically, global variable and function addresses *(e.g. &function)* are incorrect at runtime

Linker Script


```
SECTION(start)

// this jump allows for the binary loader to just jump to the
// beginning of the buffer. This must be the first thing in the binary
ENTRY_POINT(default)
    // Jump to the entry point below
    b start_asm
    // Return to calling function
    //ldr r0, =CALLSIGN
    //mov pc, lr

VARIABLE(start_callsign) .word CALLSIGN

// this is the custom header
#include "hdr.S"

FUNCTION(start_asm)
    push {r1 - r12, lr}
    bl int_disable_fiq
    bl int_disable_irq
    ldr r1, =CALLSIGN
    push {r1}
    bl start_c
    pop {r1}
    push {r0}
    bl int_enable_irq
    bl int_enable_fiq
    pop {r0}
    pop {r1 - r12, pc}
```

```
$(TARGET): $(OBJS)
    @echo "building $(TARGET)"
    @$(LD) $(LFLAGS) -o $(BINDIR)/$(TARGET).elf $(OBJS) $(OBJECTS)
    @$(OBJCOPY) $(OBJCOPYFLAGS) $(BINDIR)/$(TARGET).elf $(BINDIR)/$(TARGET).bin
```

```
LFLAGS := -march=armv7-a -O0 -m armelf_linux_eabi -T linker.ld
OBJCOPYFLAGS := -S -I elf32-littlearm -O binary
```

- The linker script (right side) shows how to manually specify the layout of a binary at link time
- Constructing a flat binary is a multistage process
  1. Create object files *(i.e. gcc –c option)*
  2. Create an *elf* file by using the *ld* command (bottom left)
  3. Create the flat binary by using *objcopy* (bottom left)
- The linker script also identifies a couple custom sections *(i.e. start and end)* this allows for an entry point to be placed at the top of the file (top left), and a identifier at the end of it

```
OUTPUT_FORMAT(elf32-littlearm)
ENTRY(default)

SECTIONS
{
        . = 0x00000000;
        . = ALIGN(4);
        .text : {
            start = .;
            *start.S.o(.start)
        }
        . = ALIGN(4);
        .text : {
                text = .;
                *(.text);
        }
        . = ALIGN(4);
        .text : {
                rodata = .;
                *(.rodata);
        }
        . = ALIGN(4);
        .text : {
                data = .;
                *(.data)
        }
        . = ALIGN(4);
        .text : {
                bss = .;
                *(.bss);
        }
        . = ALIGN(4);
        .text : {
                LC0 = .;
                *(.LC0);
        }
        . = ALIGN(4);
        .text : {
                *end.S.o(.end)
        }
        /DISCARD/ : {
                *(.ARM.attributes);
                *(.comment);
        }

}
```

Steps 2 and 3 can be collapsed but having an elf version makes debugging easier, as it can be loaded into objdump, ida pro and other debugging tools

Position Independence

- **This fixes incorrect global variable and function addresses at runtime**
  - Gaining position independence in this type of a build system is ***relatively*** easy
- Stack variables are not a concern as they will be correct at runtime
- The base address of the binary should be set to 0 in the linker script
- To resolve an address at runtime, add the link time address to the runtime base address of the binary

```
FUNCTION(gen_get_base)
        mov r1, lr
        bl 1f
        1:
        ldr r0, =1b
        sub r0, lr, r0
        mov pc, r1
```

```
void * gen_add_base(void *address) {
        return (void *)((u32_t)address + (u32_t)gen_get_base());
}

void * gen_subtract_base(void *address) {
        return (void *)((u32_t)address) - (u32_t)gen_get_base();
}
```

- An assembly function is used to obtain the base address of the binary at runtime
  1. It uses the branch with link instruction to store the PC address of the ***1*** label in the link register
  2. Loads the link time address of the ***1*** label into ***r0***
  3. Subtracts the two and returns it in ***r0***
- The C functions abstract the assembly function and add the runtime base address of the binary to any a link time address and obtain a valid runtime address

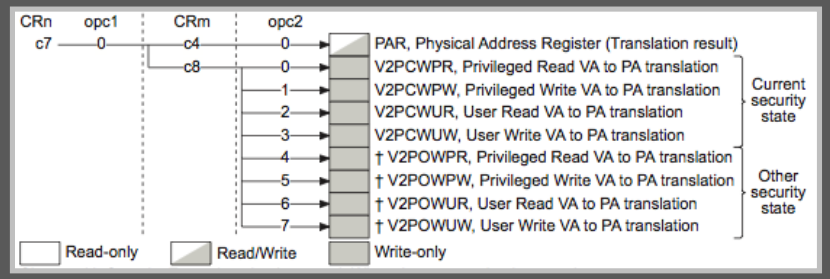
```
DBG_DEFINE_VARIABLE(mmu_dbg, DBG_LEVEL_1);

mmu_lookup_t *mmu_table;
size_t mmu_size;

mmu_paging_system_t *mmu_paging_system;
```

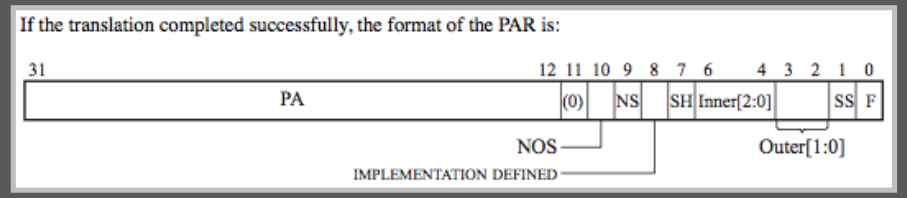
```
mmu_lookup_t **mt;
size_t *ms;
size_t i;

DBG_LOG_FUNCTION(mmu_dbg, DBG_LEVEL_3);

mt = gen_add_base(&mmu_table);
ms = gen_add_base(&mmu_size);

*mt = malloc((size / TT_SMALL_PAGE_SIZE) * sizeof(mmu_lookup_t));
```

Combining a flat binary and position independence creates a perfect environment for constructing complex OS independent applications

Brute Force Address Translation

- The ARMv7 architecture provides the ability to perform virtual to physical address translations using co-processor 15
- **This feature eliminates the need for the *loader* and microvisor to use OS specific API or knowledge to obtain translations of memory**
- This feature can be reversed to translate a physical address to a virtual address
- This feature eliminates a circular dependency associated with constructing and loading a paging system
  - Translation Table Base Registers (TTBRs) and descriptors use physical addresses
    - Translation is needed to set the TTBR correctly
    - x86 does not provide this mechanism and requires OS API to provide translations before launching a microvisor





Brute Force Address Translation

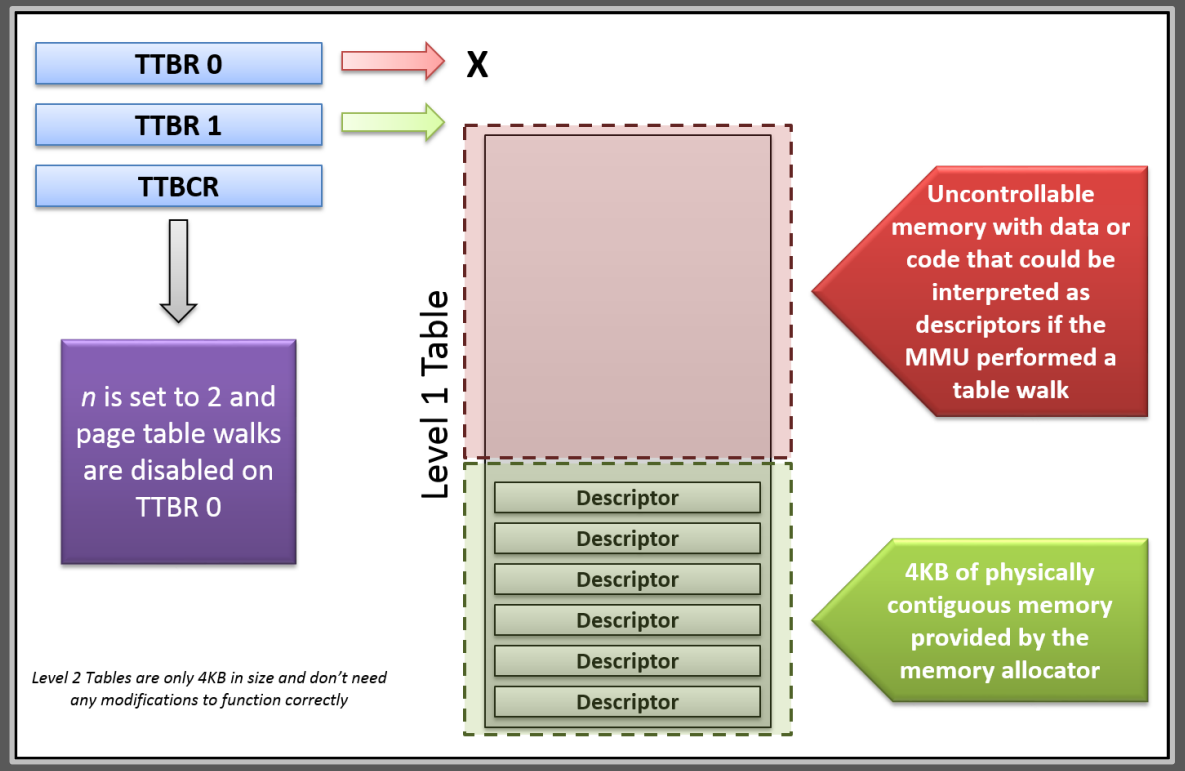
- Start at a base virtual address and loop until the physical address matches the one passed in or the end virtual address is reached

- Basic algorithm, just translate virtual addresses beginning at the start virtual address and ending at the end virtual address
  - If a virtual address translates to the physical address of interest return the translation and SUCCESS, else continue
  - If the end address is reached return FAILURE and NULL
- Co-processor 15 does not always identify the page size associated with a translation
  - Only tells if the page is a supersection or not
    - This could be used improve performance of the algorithm
  - Page size doesn't matter, the algorithm can just be generalized to assume *all* pages are small
    - *smallest page mapping available in the translation tables*

```
result_t gen_pa_to_va(tt_physical_address_t pa, tt_virtual_address_t start,
        tt_virtual_address_t end, tt_virtual_address_t *va) {

    tt_physical_address_t tmp;
    u32_t address;

    *va = start;

    address = (pa.all & ~(TT_SMALL_PAGE_SIZE - 1));
    start.all &= ~(TT_SMALL_PAGE_SIZE - 1);
    end.all &= ~(TT_SMALL_PAGE_SIZE - 1);

    for(*va = start; va->all < end.all; va->all += TT_SMALL_PAGE_SIZE) {

        gen_va_to_pa(*va, &tmp);

        if(tmp.all == address) {
            va->all |= (pa.all & (TT_SMALL_PAGE_SIZE - 1));
            return SUCCESS;
        }
    }

    va->all = (u32_t)NULL;

    return FAILURE;
}
```

¼ Size Paging Subsystem

- **A paging system allows for the microvisor to map any physical memory**
  - This can be used to communicate with peripherals or manipulate the underlying OS in a generic way
  - Ability to modify the OS's page tables
- Maintains an identity map between the OS's paging system where the microvisor was loaded
- The virtual memory address space is limited to 1GB
  - Addresses can range from 3GB to 4GB



- **Only requires 4KB of physically contiguous memory**
  - The size of a small page of memory
  - Means that any kernel memory allocator can be used *(e.g. vmalloc or kmalloc)*
- **Can map either sections or small pages**

Vector Table Modifications

- **Modifying the vector table allows the microvisor to gain control when hardware events occur before the underlying OS**
- Apple iOS and Linux mark the page that holds the vector table as read-only
  - The microvisor's paging system is used to map the physical memory that the vector table resides on to a different virtual address that is writable
- The instructions in the table are ***all*** control flow instruction of some kind so a small decoder is used to construct an absolute address of the OS's vector handlers
  - The addresses are needed so the system can reflect exceptions and interrupts into the underlying kernel as required
- The table is then replaced to direct ***all*** exceptions and interrupts to the microvisor

Vector Table Modifications


```
#define VEC_C_HANDLER(name)                                    \
        .extern vec_asm_handler_ ## name                       \

.macro VEC_ASM_HANDLER name, vector

// the address of the original handler
VARIABLE(vec_handler_\name) .word 0x0
// boolean to determine if the event was handled or not
VARIABLE(vec_handled_\name) .word 0x0
VARIABLE(vec_old_stack_\name) .word 0x0
VARIABLE(vec_new_stack_\name) .word 0x0

FUNCTION(vec_asm_handler_\name)
        // backup and switch the stack pointer
        str sp, vec_old_stack_\name
        ldr sp, vec_new_stack_\name

        // backup the registers which will in turn load the
        // gen_general_purpose_registers_t structure
        push {lr}
        // load the callsign in the the location that sp should be
        ldr lr, =CALLSIGN
        push {lr}
        push {r0 - r12}

        // put the vector into r0
        mov r0, #\vector

        // put the address of sp in r1 as it will
        // be used as the gen_general_purpose_registers_t *
        mov r1, sp

        // call the associated c function
        bl vec_dispatch_handler

        // see if the event was handled
        ldr r0, vec_handled_\name
        cmp r0, #FALSE
        bne 1f

        // it was not handled jump to the operating system handler
        pop {r0 - r12}
        add sp, $4 // space for size_t sp
        pop {lr}

        // restore the old stack pointer
        ldr sp, vec_old_stack_\name
        // branch to the operating system handler
        ldr pc, vec_handler_\name

        // it was handled return to the originator
        1:
        pop {r0 - r12}
        add sp, $4 // space for size_t sp
        pop {lr}

        // restore the old stack pointer
        ldr sp, vec_old_stack_\name
        // switch the mode to the one in spsr and return
        movs pc, lr
.endm
```

Macro used as the entry point for the exception and interrupt vectors

```
                        ; DATA XREF: reset_vector↓o ...
B               reset_vector
ON CHUNK FOR reset_vector
-------------------------------------------------------------------
LDR             PC, =undefined_instruction_vector
-------------------------------------------------------------------
LDR             PC, =supervisor_call_vector
-------------------------------------------------------------------
LDR             PC, =prefetch_abort_vector
-------------------------------------------------------------------
LDR             PC, =data_abort_vector
-------------------------------------------------------------------
LDR             PC, =reserved_vector
-------------------------------------------------------------------
LDR             PC, =irq_vector
-------------------------------------------------------------------
LDR             PC, =fiq_vector
-------------------------------------------------------------------
DCD 0x5FF00040
DCD undefined_instruction_vector ; DATA XREF: RAM:5FF00004↑r
DCD supervisor_call_vector ; DATA XREF: RAM:5FF00008↑r
DCD prefetch_abort_vector ; DATA XREF: RAM:5FF0000C↑r
DCD data_abort_vector    ; DATA XREF: RAM:5FF00010↑r
DCD reserved_vector      ; DATA XREF: RAM:5FF00014↑r
DCD irq_vector           ; DATA XREF: RAM:5FF00018↑r
DCD fiq_vector           ; DATA XREF: RAM:5FF0001C↑r
```

The vector table in the Low Level Bootloader (LLB) on the iphone 4

Vector Table Modifications

- The vector subsystem allows others, such as the dynamic linker and modules, to register at runtime and receive exceptions and interrupts
- Implements a linked list to hold registered handlers and the dispatcher just loops until the list is exhausted
- Handlers notify the dispatcher if the event should be reflected into the underlying OS or not
- Register state is passed as a parameter and allows for easy modification and inspection

```
typedef result_t (* vec_function_t)(vec_handler_t *handler, size_t *handled, gen_general_purpose_registers_t *registers);
```

```
extern result_t vec_dispatch_handler(size_t vector, gen_general_purpose_registers_t *registers);
```

```
extern result_t vec_add_handler(size_t vector, vec_function_t function, void *data);
```

```
extern result_t vec_remove_handler(vec_handler_t *handler);
```

```
#define VEC_RESET_VECTOR                 EXC_RESET_INDEX
#define VEC_UNDEFINED_INSTRUCTION_VECTOR EXC_UNDEFINED_INSTRUCTION_INDEX
#define VEC_SUPERVISOR_CALL_VECTOR       EXC_SUPERVISOR_CALL_INDEX
#define VEC_PREFETCH_ABORT_VECTOR        EXC_PREFETCH_ABORT_INDEX
#define VEC_DATA_ABORT_VECTOR            EXC_DATA_ABORT_INDEX
#define VEC_NOT_USED_VECTOR              EXC_NOT_USED_INDEX
#define VEC_INTERRUPT_VECTOR             EXC_INTERRUPT_INDEX
#define VEC_FAST_INTERRUPT_VECTOR        EXC_FAST_INTERRUPT_INDEX
```

Dynamic Linker

- **The microvisor contains a dynamic linker that allows modules to utilize kernel API and API provided by other loaded modules**
- When modules are loaded and unloaded from the microvisor their *init* and *fini* function are called
- Functionality is invoked by executing a unique undefined instruction and setting registers to specific values
  - The undefined instruction dispatcher will then pass execution to the linker and load, unload or list the currently loaded modules

```
result_t init (size_t argc, u8_t *argv[]);
result_t fini (size_t argc, u8_t *argv[]);
void init (void);
void fini (void);
```

- When a module is being loaded the linker will:
  - Allocate space in the microvisor and copy the binary from user space
  - Parse the module header and patch the import table as needed to link required API
    - If a function is not found in the microvisor the module will not be loaded and the linker will error gracefully
  - Add any export functions defined by the module to a list so other modules can utilize them
  - The modules *init* function will be called
- When unloaded the linker will:
  - Reverse the actions of the loader
- When listing is requested the linker will:
  - Pass via registers information about each module
- Module entry points follow the standard C main prototype
  - Parameters are passable via the undefined instruction exception when the module is loaded and unloaded
    - Different arguments can be passed when loaded and unloaded

Module Template


```
#include <defines.h>

#include <hdrlib/mod.h>

ENTRY_POINT(default)

VARIABLE(callsign) .word CALLSIGN

// mod_header
VARIABLE(import_header_address) .word import_header
VARIABLE(export_header_address) .word export_header
VARIABLE(storage_header_address) .word storage_header

MOD_MODULE_INFORMATION testmod

// mod_import_header
import_header:
VARIABLE(import_functions_size) .word 1
VARIABLE(import_functions_address) .word import_functions

import_functions:
GEN_IMPORT_FUNCTION test_import

// mod_export_header
export_header:
VARIABLE(export_functions_size) .word 1
VARIABLE(export_functions_address) .word export_functions

export_functions:
GEN_EXPORT_FUNCTION test_export

// mod_storage_header
storage_header:

FUNCTION(init_asm)
        push {r1 - r12, lr}
        bl init_c
        pop {r1 - r12, pc}

FUNCTION(fini_asm)
        push {r1 - r12, lr}
        bl fini_c
        pop {r1 - r12, pc}
```

Also constructed as a flat binary with a custom header. Similar to the microvisor

C functions called when loaded and unloaded

```
#include <config.h>
#include <defines.h>
#include <types.h>
#include <main.h>

DBG_DEFINE_VARIABLE(main_dbg, DBG_LEVEL_3);

result_t init_c(size_t argc, u8_t *argv[]) {

        DBG_LOG_FUNCTION(main_dbg, DBG_LEVEL_3);
        return SUCCESS;
}

result_t fini_c(void) {

        DBG_LOG_FUNCTION(main_dbg, DBG_LEVEL_3);
        return SUCCESS;
}
```

Macro used to import functions into the module at runtime

```
.macro GEN_IMPORT_FUNCTION name

1:
VARIABLE(size_\name) .word (2f - 1b)

VARIABLE(address_\name) .word 0

VARIABLE(string_\name) .asciz "\name"

FUNCTION(\name)
ldr pc, address_\name
2:

.endm
```

Macro used to export functions from the module so other modules can import them at runtime

```
.macro GEN_EXPORT_FUNCTION name

1:
VARIABLE(size_\name) .word (2f - 1b)

VARIABLE(address_\name) .word \name

VARIABLE(string_\name) .asciz "\name"
2:

.endm
```

Exposed Microvisor Kernel API


```
tt_get_fld
tt_set_fld
tt_get_sld
tt_set_sld
tt_fld_to_pa
tt_sld_to_pa
tt_pa_to_fld
tt_pa_to_sld
tt_fld_is_supersection
tt_fld_is_section
tt_fld_is_page_table
tt_sld_is_not_present
tt_ttbr_to_pa
tt_get_ttbr0
tt_get_ttbr1
tt_get_ttbcr
tt_set_ttbr0
tt_set_ttbr1
tt_set_ttbcr

uart_getc
wdt_init
wdt_fini

uart_init
uart_fini
uart_set_clock
uart_write
uart_read
uart_putc
uart_getc
wdt_init
wdt_fini
wdt_interrupt_disable
wdt_interrupt_enable
wdt_reset_disable
wdt_get_reset_data
wdt_set_reset_data
wdt_get_interrupt_data
wdt_set_interrupt_data
wdt_get_count
wdt_set_count

memcpy
memset
strlen
memcmp
reverse
itoa
atoi
isspace
isdigit

bpa_flush_entire_branch_predictor_array
bpa_flush_mva_branch_predictor_array
cac_get_csselr
cac_set_csselr
cac_get_clidr
cac_get_ccsidr
cac_invalidate_mva_to_pou_instruction_cache
cac_flush_entire_instruction_cache
cac_clean_mva_to_poc_data_cache
cac_flush_mva_to_poc_data_cache
cac_flush_entire_data_cache

uart_init
uart_fini
uart_set_clock
uart_write
uart_read
uart_putc

mas_alloc
mas_free
mas_get_debug_level
mas_set_debug_level
mmu_lookup_va
mmu_lookup_pa
mmu_switch_paging_system
mmu_get_paging_system
mmu_map
mmu_unmap
mmu_get_debug_level
mmu_set_debug_level
vec_add_handler
vec_lookup_handler
vec_remove_handler
vec_get_debug_level
vec_set_debug_level
ldr_add_module
ldr_remove_module
ldr_add_function
ldr_lookup_function
ldr_remove_function
ldr_get_debug_level
ldr_set_debug_level
wdt_get_reset_data
wdt_set_reset_data
wdt_get_interrupt_data
wdt_set_interrupt_data
wdt_get_count
wdt_set_count

cpuid_get_midr
gen_get_sp
gen_set_sp
gen_get_cpsr
gen_set_cpsr
gen_get_spsr
gen_set_spsr
int_disable_irq
int_enable_irq
int_disable_fiq
int_enable_fiq
flt_get_dfsr
flt_set_dfsr
flt_get_dfar
flt_set_dfar
flt_get_ifar
flt_set_ifar
gen_pa_to_va
gen_va_to_pa
gen_get_par
gen_privileged_read_translation
gen_privileged_write_translation
gen_user_read_translation
gen_user_write_translation
gen_get_sctlr
gen_set_sctlr
tlb_get_tlbtr
tlb_invalidate_entire_unified_tlb
tlb_invalidate_mva_unified_tlb

ser_init
ser_fini
ser_write
ser_read
ser_print
memcpy
memset
strlen
memcmp
reverse
itoa
atoi
isspace
isdigit

wdt_interrupt_disable
wdt_interrupt_enable
wdt_reset_disable
```

Loaders Exposed

Explanation of how loading is accomplished on both Apple iOS and Google Android.

Apple iOS Loader (Jailbroken)

- Currently Apple iOS does *not* provide the ability to install kernel modules even when the device is jailbroken
  - It uses a cached version of the kernel that has all the modules loaded, thus eliminating the need for loadable module support
- The *loader* performs the following actions in user space
  - Allocates non-executable memory in kernel space
  - Copies the core microvisor into it
  - Initialized the serial port to 115200 baud rate
  - Patches a function pointer in the system call table
  - Makes a system call to execute the function it patched into the system call table with supervisor privilege *(i.e. kernel mode)*
- While Mach system calls were used to accomplish the task it is analogous to using *mmap* and *"/dev/kmem"* on Linux
  - The basic requirement is just the ability to perform arbitrary kernel memory reads and writes
  - Specific functions used include *task_for_pid, mach_self_task, vm_read_overwrite, vm_write and vm_allocate*

Jailbreak code from Comex was referenced

Apple iOS Loader (Jailbroken)

- Once executing in kernel space the following actions are performed before jumping to the microvisor
  - The correct translation table base address is obtained and a brute force physical to virtual address translation is performed
  - The level 1 OS page table is then walked to find the descriptors associated with the memory allocated in user space for the microvisor
    - This involves performing brute force translations on level 2 tables when required
  - The descriptors are then modified to make them executable and writable (neuter XN)
- When the microvisor finishes initialization the loader returns to user space and does the following
  - Patches the system call table again to restore the original entry
  - Drops into an infinite loop to maintain the serial port settings (can be removed if not debugging with the UART)

No kernel API were harmed or used to perform the required actions, only architectural features were leveraged

Apple iOS Loader (Jailbroken)

```
de 48 89 c7 e8 73
00 00 00 48 8d 54
48 89 c7 e8 c0 00
b8 ed c7 7e de 48
89 b9 01 00 8b 00
48 89 c3 bf 10 2c
48 89 c7 e8 93 67
```

```
Kirk-Swidowskis-iPhone:/ root# /ios_sysldr /system.bin
        ------------------------------
        -    v*v  0xde7ec7ed  v*v    -
        -    *^*  iOS sysldr  *^*    -
        ------------------------------
[+] serial port opened
[+] restricted usage of the serial port
[+] cleared the non blocking state on the serial port
[%] set input baud rate to: 115200
[%] set output baud rate to: 115200
[+] set attributes to take effect immediately
[+] serial port initialized
[%] kernel loaded at 80001000
[%] file: /system.bin
[+] loaded the flat binary file
[%] flat binary loaded at kernel address: 0xd41c7000
[%] flat binary size: 58992
[+] patched the flat binary file
0:      df 02 00 ea ed c7 7e de
8:      14 00 00 00 2c 00 00 00
10:     84 0b 00 00 00 70 1c d4
18:     22 22 22 22 00 00 40 00
20:     01 00 00 00 00 00 00 00
28:     2c 00 00 00 77 00 00 00
30:     34 00 00 00 17 00 00 00
38:     dc c4 00 00 63 70 75 69
40:     64 5f 67 65 74 5f 6d 69
48:     64 72 00 13 00 00 00 24
50:     c5 00 00 67 65 6e 5f 67
58:     65 74 5f 73 70 00 13 00
60:     00 00 2c c5 00 00 67 65
68:     6e 5f 73 65 74 5f 73 70
70:     00 15 00 00 00 34 c5 00
78:     00 67 65 6e 5f 67 65
                ...
[+] got nsysent
[%] nsysent: 0x000001b7
[+] got space for sysent
[+] read the sysent table
[+] backed up the old sysent
[+] replaced the sysent
[+] wrote the sysent table
[%] system_call address: 0x00002af8
[%] making syscall in : 3 2 1
[+] syscall returned success: 0x00000000
[+] restoring the old sysent
[+] wrote the sysent table
[+] crossed boundary and launched
[+] entering infinite loop press CTRL^Z to return to the prompt
```

Android Loader (rooted)

- Constructed as a kernel module
- Disables secondary processor core on the Samsung SIII
  - This may not be required but makes debugging easier
- Propagates page mappings throughout the systems address spaces
  - This is required because the microvisor hooks the vector table and needs to make sure that its memory is mapped in all the address spaces
    - If Linux used *ttbr1* correctly this wouldn't be required
- Reads the microvisor kernel into memory from a file
  - **Disclaimer:** Don't do this in legit kernel modules as the Linux community will be very upset (something about separation of policy and mechanism)
  - Using the File I/O API from within the Kernel is a no-no!
    - Linux kernel (No = File I/O, Yes = Process Creation) Windows Kernel (No = Process Creation, Yes = File I/O)

SYS$OUTPUT blog and code referenced

Modules & Management

Existing modules and the unprivileged user space management applications.

Management Applications

- Three user space applications can be used to install, remove and list the modules *(i.e. ios_insmod, ios_rmmod, ios_lsmod)*
- All the programs are very basic and just throw undefined instruction exceptions to communicate with the underlying microvisor
- Microvisor catches the exception and returns execution to the originating code without notifying the underlying OS kernel
- They can be quickly ported to other ARM OSs with little effort
  - Ported from iOS to Android in less than an hour

```
Kirk-Swidowskis-iPhone:~ root# /ios_insmod /wdtmod.bin
        ------------------------------
        -    v*v  0xde7ec7ed  v*v    -
        -    *^*  iOS insmod  *^*    -
        ------------------------------
[%] file: /wdtmod.bin
[+] module successfully added
```

```
Kirk-Swidowskis-iPhone:~ root# /ios_lsmod
        ------------------------------
        -    v*v  0xde7ec7ed  v*v    -
        -    *^*  iOS  lsmod  *^*    -
        ------------------------------
name            import  export  init    fini
wdtmod          0x001f  0x0194  0x019c  0x01a8
[+] module successfully listed
```

```
Kirk-Swidowskis-iPhone:~ root# /ios_rmmod wdtmod
        ------------------------------
        -    v*v  0xde7ec7ed  v*v    -
        -    *^*  iOS  rmmod  *^*    -
        ------------------------------
[+] module successfully removed
Kirk-Swidowskis-iPhone:~ root#
```

Anything that can induce the required undefined instruction exception can manage modules *(i.e. the loader could automatically add additional modules at installation time)*

Modules

- Breakpoint Module
  - Set breakpoints and singlestep any underlying code including kernel API, FIQ handlers, IRQ handlers, etc.
  - Supports both ARM and Thumb mode
  - Exports API to register and unregister handlers
- Light Weight Shadow Paging Module
  - Monitor any MMIO on the system
  - Exports API to register and unregister handlers
  - This is currently used to aid in reverse engineering of hardware peripherals
- Linux Module
  - Finds and maintains links to kernel API
  - Exports API to call many useful kernel functions, such as:
    - *kmalloc*
    - *kfree*
    - *printk*
    - *call_usermodehelper_setup*
    - *call_usermodehelper_exec*
    - *schedule*
    - *create_proc_entry*
    - *remove_proc_entry*

Modules

- SOC Modules
  - MSM8960 (Samsung Galsxy SIII) [Qualcomm SoC]
  - S5L8930 (Apple iPhone 4) [Samsung SoC]
  - Will only load on the correct hardware (uses something analogous to CPUID on x86)
  - Exposes SoC specific API (*e.g. UART, watchdog*)
- Cryptography Module
  - Provides API for two implementations of AES-128 (s-box and table based)
  - Provides API for SHA-1
- PL192 Module
  - Exposes API to interact with the PrimeCell PL192 Vectored Interrupt Controller
    - Reflect software interrupts into the underlying OS
    - Disable Interrupts
    - Set if an interrupt is ran in IRQ or FIQ mode

Modules

- Watchdog Module
  - Disables the watchdog timer and intermittently checks to make sure it has not been re-enabled by the underlying OS.
  - The watchdog timer is a custom format and does not match up to any SoC datasheets I had access to.
  - Development of this module involved:
    - Static analysis of the LLB, iBoot and the iOS kernel.
    - Dynamic analysis using the shadow paging and breakpoint modules.
- Paging Module
  - Print out all page table descriptors that meet a set of attributes
  - It was used to find the base addresses of all potential MMIO on the iPhone 4
  - It was later verified by a modified python script that parsed the DeviceTree.img3 file in the IPSW package.
    - The version online did not work on newer iOS devices and needed to be updated
    - The script provided the entire MMIO, VIC and GPIO layout among other things

Debugging Facilities

Software debugging facilities and custom hardware cables.

Apple iPhone 4 Debugging

- UART 0 on the iPhone 4 resides at physical address 0x82500000
- The virtual address in kernel space was found by using co-processor 15's ability to perform memory address translations (unique brute force approach)
- Knowing that the iPhone 4 (S5L8930) is a Samsung SoC another Samsung SoC's datasheet was used to develop a UART library (the register layout is the same)
- The UART on older iDevices is exposed through the bottom connector (pin 12 is TX and 13 is RX)

- A 470K ohm pull down resister is connected across pin 1 and 21 to enable the UART.

Referenced "Targeting the iOS Kernel" by Stefan Esser for the hardware

Samsung Galaxy SIII Debugging

```
de 48 89 c7 e8 73
00 00 00 48 8d 54
48 89 c7 e8 c0 00
b8 ed c7 7e de 48
89 b9 01 00 8b 00
48 89 c3 bf 10 2c
48 89 c7 e8 93 67
```

- A 525K ohm pull down resister is connected across the ID and GND pins of the USB port to enable the UART
- Works with *minicom* and other serial console applications

- UART 0 on the Galaxy SIII resides at physical address 0x16440000
- The virtual address in kernel space was found by using co-processor 15's ability to perform memory address translations (unique brute force approach)
- Knowing that the USA version of the Samsung Galaxy SIII is built with a Qualcomm SoC (msm8960) another Qualcomm SoC's datasheet was used to develop a UART library as the register layout was similar
- A potentiometer was used to find the correct resistance level
  - Multiple resistance levels enable the UART (they output different messages)

Construction & Layout

Repository layout and build commands are provided

Downloading & Building

- **VERTIGO consists of a number of *git* repositories**
- **The easiest way to checkout the code is to type *"git clone https://github.com/de7ec7ed/vertigo.git"* from a terminal**
  - This repository does not contain any code other than a Makefile
    - To checkout the repositories for VERTIGO type *"make clone"*
- ***Once everything is checked out type "make PREFIX=arm-linux-gnu-"***
  - *arm-linux-gnu-* is the prefix of your tool chain (e.g. arm-linux-gnu-gcc)
    - Ubuntu this can be installed by typing *"apt-get install gcc-arm-linux-gnueabi"*
    - Fedora this can be installed by typing *"yum install gcc-arm-linux-gnu.x86_64"*

Repository Layout & Binary Sizes

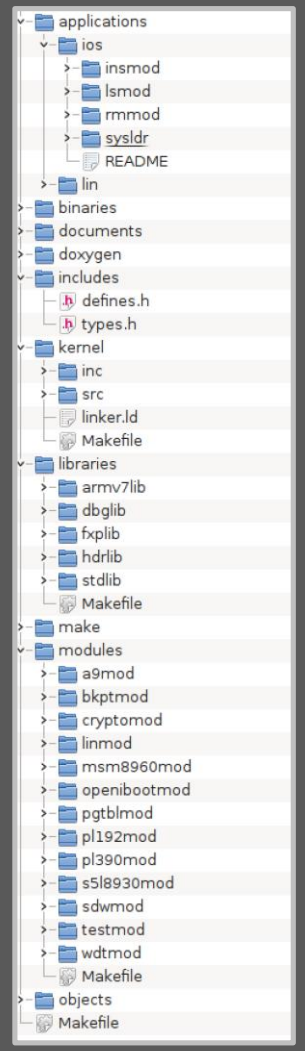
OS specific applications to install and work with modules and the microvisor.

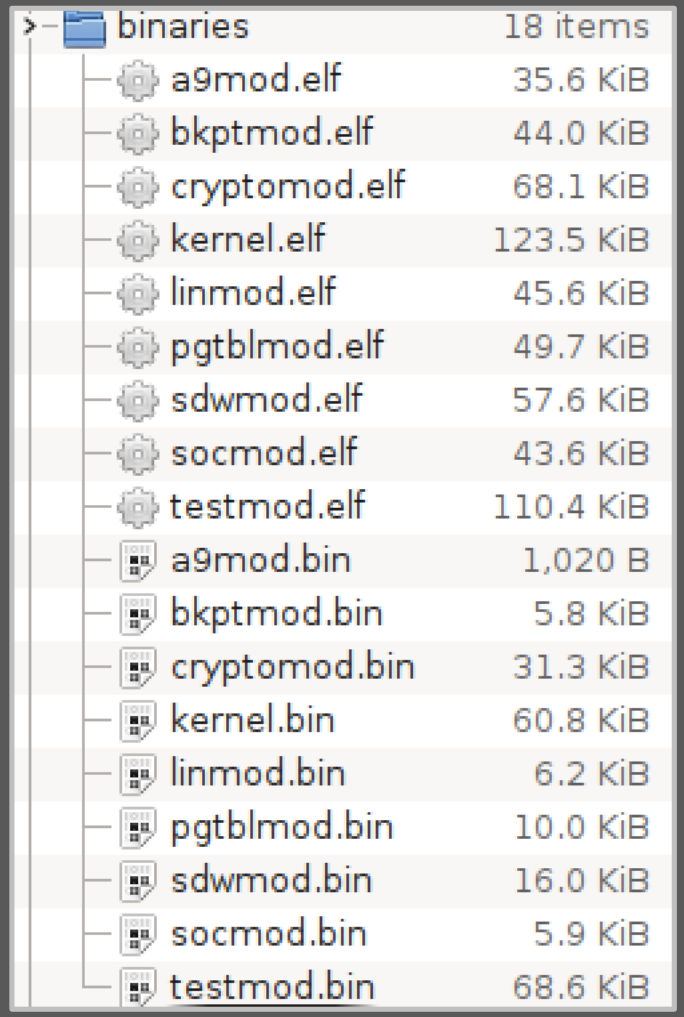
Contains a copy of the includes from the libraries, kernel and modules

The kernel

Statically linked libraries that are built into the kernel

Modules that can be linked into the kernel at runtime





ELF versions are useful for debugging (subtract the runtime base to locate things in the ELF file)

Kernel is only 60.8 KB (includes 6.9 KB of strings most are for debugging)

Average module is 10.88 KB

Questions

Kirk Swidowski

mail *kirk@swidowski.com*
web *www.de7ec7ed.com*
twitter *de7ec7ed*
github *github.com/de7ec7ed*