SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM

Ahmed M Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, Peng Ning

Samsung KNOX R&D, Samsung Research America
Motivation

• Operating system kernels still suffer from exploits
  - CVE-20XX-XXXX

• Security tools
  - Monitor and protect the kernel
  - May have large code base
  - May introduce vulnerabilities

• Isolation is a key requirement for hosting security tools
Motivation (cont.)

• Previous approaches
  - Host security tools in hypervisors and hardware security features
    o Designed with different objectives
    o Increase TCB size, increase attack surface
  - Hypervisors and hardware security features may be compromised
    o Due to the vulnerabilities introduced by security tools
    o Worse than kernel being compromised
    o Undermine the overall system security
Secure Kernel-level Execution Environment

- Lightweight *in-kernel* isolation
  - Run at the same privilege level as kernel
  - Safe from potential kernel vulnerabilities
  - No requirement of active involvement from higher privileged layers

- Ability to inspect kernel state
  - Full access to entire kernel memory
  - Event driven monitoring

- Secure context-switching
  - Entry point exposed to the kernel yet secure from attacks
Scope

• Assumptions
  - The system is booted securely
  - The kernel code is validated and protected
    o No kernel code injection
    o Valid assumption using existing techniques (e.g., W^X, DEP, PXN)

• Threat model
  - All data attacks against the kernel are considered
    o Including code-reuse attacks and non-control data modification
  - SKEE guarantees a fully compromised kernel cannot:
    o Revoke the isolation
    o Compromise the context switching
SKEE Design

• Basic idea
  - A new self-protected virtual address space

• Both address spaces are initialized at boot up time
  - Secure boot is required

• Three basic requirements
  - Isolation
  - Secure context switching
  - Kernel monitoring and protection
Isolation

• Create a protected address space
  - Instrument the kernel translation tables
    o Carve out SKEE’s physical memory range

• Restrict kernel access to the MMU
  - Revoke write access to kernel translation tables
    o Enforce W^X protection, DEP and PXN of user code
  - Remove op codes of certain instructions from kernel code
    o E.g., set TTBR value, disable the MMU
  - The kernel is forced to request MMU operations from SKEE
    o Inspected to guarantee the isolation
Secure Context Switching

• Atomic ➔ Execution never returns to kernel while SKEE is accessible
  - Potential attacks
    o Jump to the middle of the switch gate
    o Interrupt the switching logic execution

• Deterministic ➔ The switch gate shows same behavior regardless of:
  - Current system state
  - Input parameters

• Exclusive ➔ The switch gate is the only entry point to SKEE
Secure Switching on 32-bit ARMv7

• Memory management in ARMv7
  - Two translation table base registers: TTBR0 & TTBR1
    o TTBR holds the page table base, the same with CR3 in x86

• Challenge
  - Cannot load values into TTBR0 & TTBR1 in kernel directly
    o Compromise the isolation by loading unverified page tables

• Solution:
  - Use dedicated registers for the kernel and SKEE
    o Valid technical assumption (Android linux kernel only uses TTBR0)
  - Context switching is done by updating TTBCR.N
    o No direct value loading to TTBR
    o Non-zero value maps SKEE, zero value maps the kernel
Secure Switching on 32-bit ARMv7 (Cont.)
ARMv7 Switch Gate

- Lines 2-5
  - Disable interrupts
- Lines 7-10
  - Load TTBCR
- Lines 12 and 13
  - Invalidate the TLB
- Line 15
  - Jumps to SKEE
- Exit in reverse order
Atomic Switch Gate

- Control flow change
  - Branching
  - Exceptions
  - Interrupts

- Threat
  - Skip interrupt disable
  - Use TLB cached code

- Solution
  - Instrument the interrupt handler
    - Check TTBCR.N
    - Crash on non-zero (SKEE is exposed)
Deterministic and Exclusive Switch Gate

- **Deterministic**
  - No reliance on input

- **Exclusive**
  - No TTBR0, TTBR1 or TTBCR instructions exist in the kernel code

```
/* Start of the SKEE Entry Gate */
mrs r0, cpsr          // Read the status register
push {r0}             // Save the status register value
orr r0, r0, #0x1c0    // Set the mask interrupts bits
msr cpsr, r0          // load the modified value
mov r0, #0x11          // Synchronization barrier
mcr p15, 0, r0, c2, c0, 2 // Modify the TTBCR to activate SKEE
isb                    // TLB invalidate
mcr p15, 0, r0, c8, c7, 0 // Jump to SKEE entry point
/* End of the SKEE Entry Gate */

/* Start of the SKEE Exit Gate */
mov r0, #0
isb
mcr p15, 0, r0, c2, c0, 2 // Modify the TTBCR to deactivate SKEE
isb
mcr p15, 0, r0, c8, c7, 0 // TLB invalidate
isb
pop {r0}              // Reload status register value
msr cpsr, r0          // Restore the original status register
bl kernel_entry       // Jump back to the kernel
/* End of the SKEE Exit Gate */
```
Secure Switching on 64-bit ARmv8

• Memory management in 64-bit ARmv8
  - Different virtual memory subranges for TTBR0 and TTBR1
    o TTBR1: High address range; Typically used by kernel
    o TTBR0: Low address range; Typically used by user space

• Challenge
  - TTBR0 and TTBR1 map mutually exclusive memory ranges
  - Cannot dedicate either registers to SKEE

• Solution
  - SKEE shares TTBR1 with the kernel
  - Entry gate uses a special encoding
    o the Zero register (XZR)
    o Guarantee deterministic change of TTBR1
Secure Switching on 64-bit ARMv8 (cont.)

• The presence of physical address 0x0
  - Provided by the hardware as a real physical address
    o Don’t need hypervisor support
  - Provided by the virtualization layer as an intermediate physical address (IPA)
    o Need hypervisor to remap IPA0 x0 to SKEE
    o Don’t require any “runtime” hypervisor involvements
ARMv8 Entry Gate

- Lines 2-4
  - Disable interrupts
- Lines 6-10
  - Save exiting TTBR1
  - Load TTBR1 using XZR
- Lines 12 and 13
  - Invalidate the TLB
- Lines 15 and 16
  - Jump to SKEE
ARMv8 Entry Gate

• Atomic
  - Kernel cannot skip interrupt disable step
  - Jump to SKEE uses absolute address

• Deterministic

• Exclusive

```c
1 /* Start of the SKEE Entry Gate */
2 mrs x0, DAIF        // Read interrupt mask bits
3 str x0, [sp, #-8]]! // Save interrupt mask bits
4 msr DAIFset, 0x3    // Mask all interrupts
5 mrs x0, ttbr1_el1   // Read existing TTBR1 value
6 str x0, [sp, #-8]]! // Save existing TTBR1 value
7 mrs ttbr1_el1, xzr  // Load the value Zero to TTBR1
8 isb
9 tlbi vmallel1       // Invalidate the TLB
10 isb
11 adr x0, skee_entry // Jump to SKEE entry point
12 br x0
13 /* End of the SKEE Entry Gate */
```
ARMv8 Exit Gate

- Lines 2-5
  - Memory padding
  - Pushing line 11 to the isolated page boundary
- Line 7
  - Mask interrupts
- Lines 9-11
  - Reload kernel’s TTBR1
- Lines 15-17
  - Invalidate the TLB
- Lines 20-23
  - Restore interrupts and return to kernel

```assembly
/* Start of the SKEE Exit Gate */

nop //no operation
nop // Fill the page with no operations to
nop // align the last instruction with the
nop // bottom of the isolated page boundry

msr DAIFset, 0x3 // Mask all interrupts

ldr x0, [sp, #8]! // Reload kernel TTBR1 value
dsb sy
msr ttbr1_el1, x0 // Restore TTBR1 to kernel value

/*------------------Isolated Page Boundary-------------------------*/

isb

tlbi vmalle1 // Invalidate the TLB

isb

ldr x0, [sp, #8]! // Reload interrupts mask bits
msr DAIF, x0 // Restore interrupts mask bits register
ret

/* End of the SKEE Exit Gate */
```
ARMv8 Exit Gate

- **Line 11**
  - Load ttbr1 from stack
  - Can be exploited by attackers

```c
1 /* Start of the SKEE Exit Gate */
2 nop           // no operation
3 nop           // Fill the page with no operations to
4 nop           // align the last instruction with the
5 nop           // bottom of the isolated page boundary
6 msr          DAIFset, 0x3    // Mask all interrupts
7 ldr x0, [sp, #8]! // Reload kernel TTBR1 value
8 dsb sy
9 mrs ttbr1 ell1, x0     // Restore TTBR1 to kernel value
10 /*---------------------Isolated Page Boundary----------------------*/
11 isb
12 tlbi vmalle1         // Invalidate the TLB
13 isb
14 ldr x0, [sp, #8]!    // Reload interrupts mask bits
15 msr DAIF, x0         // Restore interrupts mask bits register
16 ret
17 /* End of the SKEE Exit Gate */
```
ARMv8 Exit Gate

- Page on top
  - Only accessible to SKEE

- Page on bottom
  - Accessible to both SKEE and kernel

```c
/* Start of the SKEE Exit Gate */

nop //no operation
nop // Fill the page with no operations to
nop // align the last instruction with the
nop // bottom of the isolated page boundry
msr DAIFset, 0x3 // Mask all interrupts
ldr x0, [sp, #8]! // Reload kernel TTBR1 value
dsb sy
msr ttbr1_el1, x0 // Restore TTBR1 to kernel value

/*-------------------Isolated Page Boundry-------------------*/
isb
tlbi vmalle1 // Invalidate the TLB
isb

ldr x0, [sp, #8]! // Reload interrupts mask bits
msr DAIF, x0 // Restore interrupts mask bits register
ret

/* End of the SKEE Exit Gate */
```
Fast Secure Switching using ASID

• Entire TLB invalidation
  - Potential performance overhead

• Using a dedicated ASID for SKEE
  - Non-global mapping of SKEE memory
  - TLB entries will only be associated with a particular ASID
  - No need to flush the TLB on every context switch

• Global mapping of the switch gate
  - Accessible to both the kernel and SKEE
Kernel Monitoring and Protection

- Control page table
  - Make sure the page table is properly set up, with W^X, DEP and PXN on user

- Replace the MMU instruction with hooks to SKEE
  - The hook will trap to SKEE
  - SKEE will check each operation

- For hosting security tools
  - Trap critical kernel events
  - Inspect kernel memory
Performance

- Secure context switching
  - No TLB invalidation → ASID is used

<table>
<thead>
<tr>
<th>Processor</th>
<th>Average Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMv7</td>
<td>868</td>
</tr>
<tr>
<td>ARMv7 (No TLB invalidation)</td>
<td>550</td>
</tr>
<tr>
<td>ARMv8</td>
<td>813</td>
</tr>
<tr>
<td>ARMv8 (No TLB invalidation)</td>
<td>284</td>
</tr>
</tbody>
</table>
Performance (cont.)

- Benchmark performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original</th>
<th>SKEE</th>
<th>Degradation (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF-Bench</td>
<td>30933</td>
<td>29035</td>
<td>6.14%</td>
</tr>
<tr>
<td>Smartbench 2012</td>
<td>5061</td>
<td>5002</td>
<td>1.17%</td>
</tr>
<tr>
<td>Linpack</td>
<td>718</td>
<td>739</td>
<td>-2.93%</td>
</tr>
<tr>
<td>Quadrant</td>
<td>12893</td>
<td>12552</td>
<td>2.65%</td>
</tr>
<tr>
<td>Antutu v5.7</td>
<td>35576</td>
<td>34761</td>
<td>2.29%</td>
</tr>
<tr>
<td>Vellamo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Browser</td>
<td>2465</td>
<td>2500</td>
<td>-1.42%</td>
</tr>
<tr>
<td>Metal</td>
<td>1077</td>
<td>1071</td>
<td>0.56%</td>
</tr>
<tr>
<td>Geekbench</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single Core</td>
<td>1083</td>
<td>966</td>
<td>10.8%</td>
</tr>
<tr>
<td>Multi Core</td>
<td>3281</td>
<td>2747</td>
<td>16.28%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original</th>
<th>SKEE</th>
<th>Degradation (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF-Bench</td>
<td>75641</td>
<td>66741</td>
<td>11.77%</td>
</tr>
<tr>
<td>Smartbench 2012</td>
<td>14030</td>
<td>13377</td>
<td>4.65%</td>
</tr>
<tr>
<td>Linpack</td>
<td>1904</td>
<td>1874</td>
<td>1.58%</td>
</tr>
<tr>
<td>Quadrant</td>
<td>36891</td>
<td>35595</td>
<td>3.51%</td>
</tr>
<tr>
<td>Antutu v5.7</td>
<td>66193</td>
<td>67223</td>
<td>-1.56%</td>
</tr>
<tr>
<td>Vellamo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Browser</td>
<td>3690</td>
<td>3141</td>
<td>14.88%</td>
</tr>
<tr>
<td>Metal</td>
<td>2650</td>
<td>2540</td>
<td>4.15%</td>
</tr>
<tr>
<td>Geekbench</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single Core</td>
<td>1453</td>
<td>1235</td>
<td>15.00%</td>
</tr>
<tr>
<td>Multi Core</td>
<td>4585</td>
<td>4288</td>
<td>6.48%</td>
</tr>
</tbody>
</table>

2/24/2016
Thank you