TRUSTSHADOW: SECURE EXECUTION OF UNMODIFIED APPLICATIONS WITH ARM TRUSTZONE

14.11.2018

Florian Olschewski

OUTLINE

1) Introduction
2) Trustzone
3) Threat Model
4) Overview
5) Runtime System
6) Implementation
7) Evaluation
8) Future Work

1) INTRODUCTION

- Rapid evolution of IOT-Devices
- Problem: compromised OS
  - Leak of sensitive Data
- **TrustShadow(TS)**: shields applications from untrusted OS
- **TS** uses ARM-Trustzone
  - *Normal world* →OS
  - *Secure world* → TEE : critical application
- Secure world is managed by a leightweight runtime system(RTS)
  - Forwards system calls + verifies responses

2) TRUSTZONE - ARCHITECTURE

- *Partition* of SoC- hardware + software in secure and normal world
- Processor can enter normal and secure state
  - Normal state: access to resources in normal world
  - Secure state: access to all resources
- To check permissions: Non-Secure bit
- Monitor mode software to switch between the worlds

2) TRUSTZONE - ADDRESS SPACE CONTROLLER + MEMORY MANAGEMENT UNIT(MMU)

- Set-up security **access permissions** for address regions
- **Controls data transfer** between processor and Dynamic Memory Controller
  - NS-bit must equal the security setting of memory region

- MMU: Translation of virtual to physical addresses
- Memory splitted in 2 worlds → 2 MMU's for **independent** memory mapping
  - *Normal world*: only access to memory in non-secure state
  - *Secure world*: access to both memory states by *tuning NS-bit*

3) THREAT MODEL

- Shielding applications from completely hostile OS
  - Memory disclosure
  - Code injection attacks
  - Change program behavior
  - Side channel attacks (e.g. observe page fault pattern)
- No prevention for
  - DoS-attacks: OS refuses to boot / decline time slices for a process
  - Side channel like timing and power analysis

4) OVERVIEW

- Trusted application:
  - Customized system call:
  - „zombie“ HAP: normal world, never scheduled „shadow“ HAP: secure world, ran by TrustShadow
- RTS forwards exceptions to Linux
- Data structures task_shared / task_private

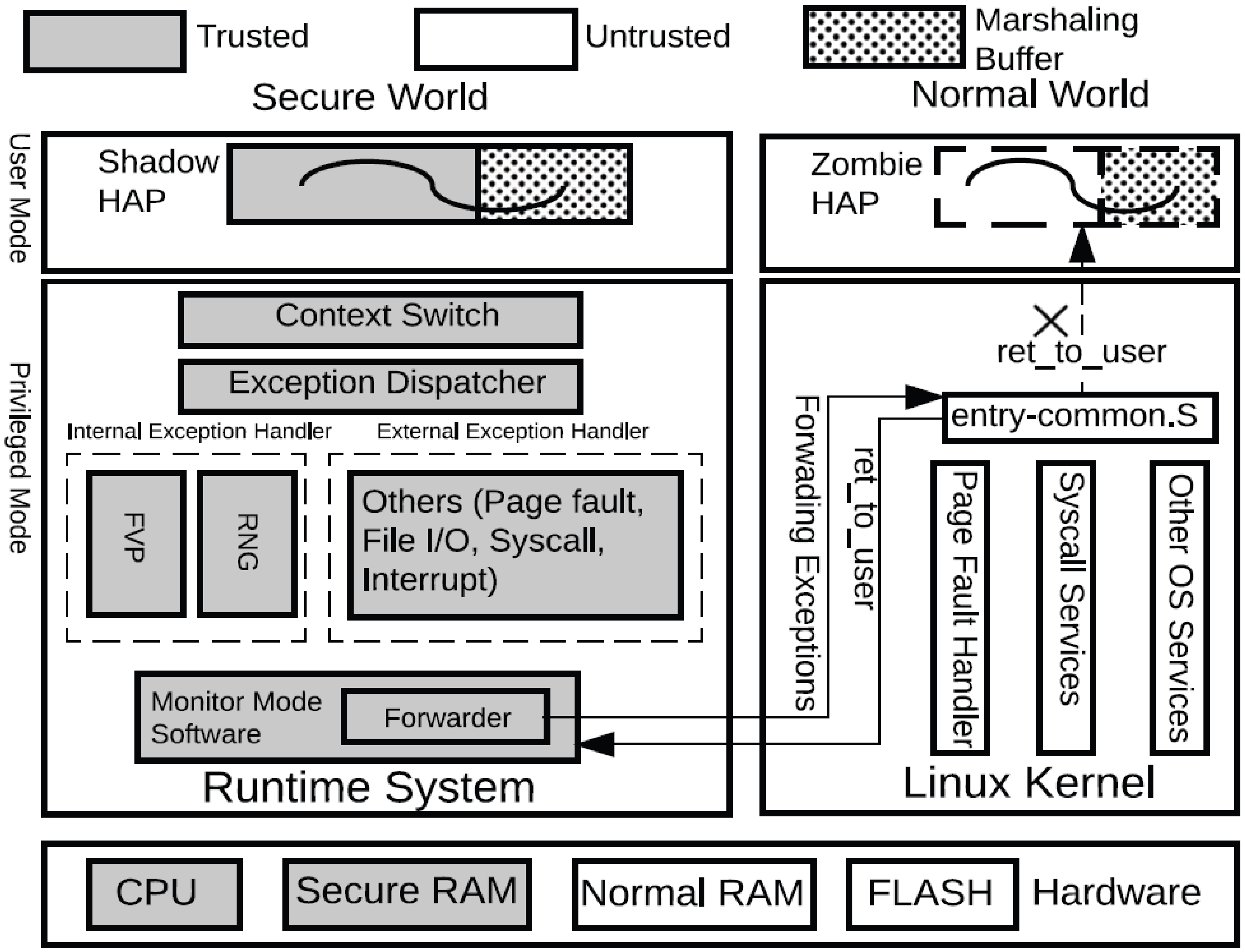


Figure 1: Architecture of TrustShadow

5) RTS - MEMORY MANAGEMENT

- 3 partitions of physical memory:
  - *Non-secure*: **ZONE_NORMAL** – Linux OS
  - *Secure*: **ZONE_TZ_RT** – for runtime system
    ZONE_TZ_APP – shadow-HAP‘s
- Virtual memory:
  - user/kernel memory split of secure world equals Linux
    → execution of legacy code in secure world
  - RTS maps itself to ZONT_TZ_RT
  - maps memory holding Linux in the virtual address space
    →efficiently locate shared Data from OS

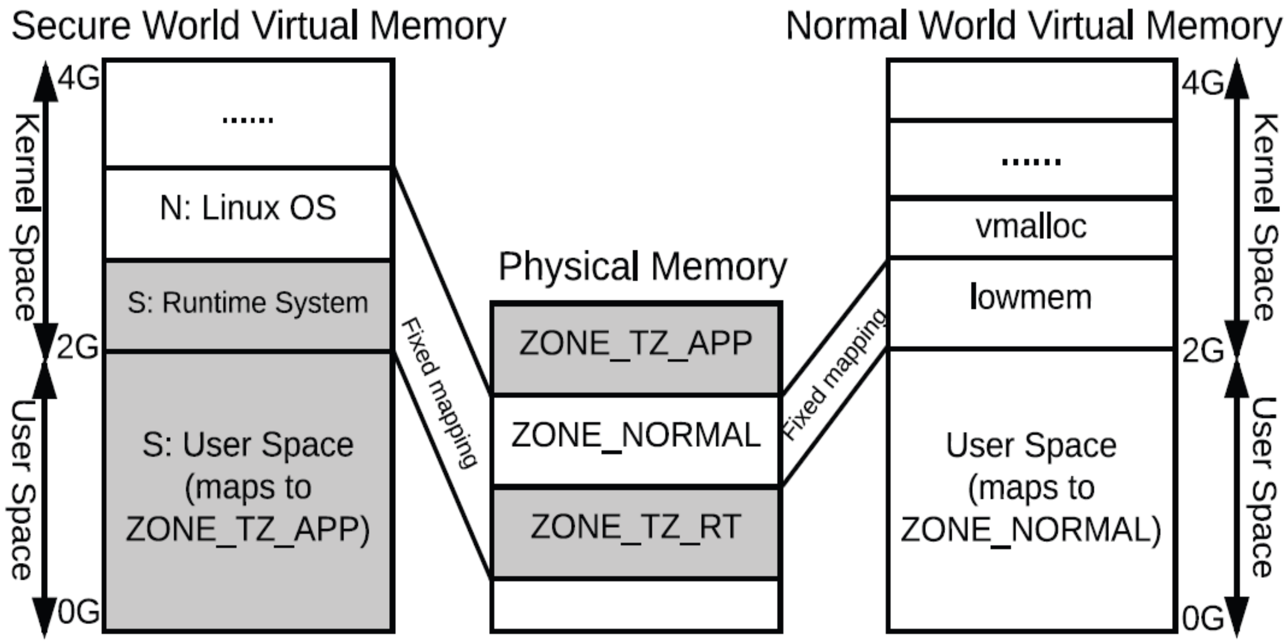


Figure2: physical + virtual memory layout

5) RTS - FORWARDING EXCEPTIONS

Exception handling of ARM-Processors:

1. Pc points exception vector table
2. store previous cpsr to spsr
   - Every processor mode has its own spsr register (banked Register)
3. Setting cpsr to indicate the target mode
   - Spsr reveals information of pre-exception processor mode

current program status register (cpsr)

saved program status register (spsr)

Reproduction by RTS (e.g. svc)

1. Set spsr in monitor mode to represent target mode (svc)
2. Switch to target mode (svc) + set it's spsr to represent User-Mode
3. Switch back to monitor mode
4. Issue movs instruction
   - Jump to target exception handler
   - Copy spsr from current mode in cpsr

→ OS catches exception at correct address + in the right mode (svc, step1)

→ Spsr indicates: exception comes from user mode (step 2)

5) RTS - HANDLING PAGE FAULT

- Exception by MMU → no page table entry for accessed memory
- OS maintains page tables
- RTS maintains own page table in secure world
  - Uses Linux page fault handler for updating
  - For TS, the Linux handler was modified: it stores the updated entry value to *task_shared*

Basic Page Table update:

- Anonymous memory
  - RTS verifies that the provided entry of task_shared is within ZONE_TZ_APP
  - RTS duplicates page table entry

5) RTS - HANDLING PAGE FAULT

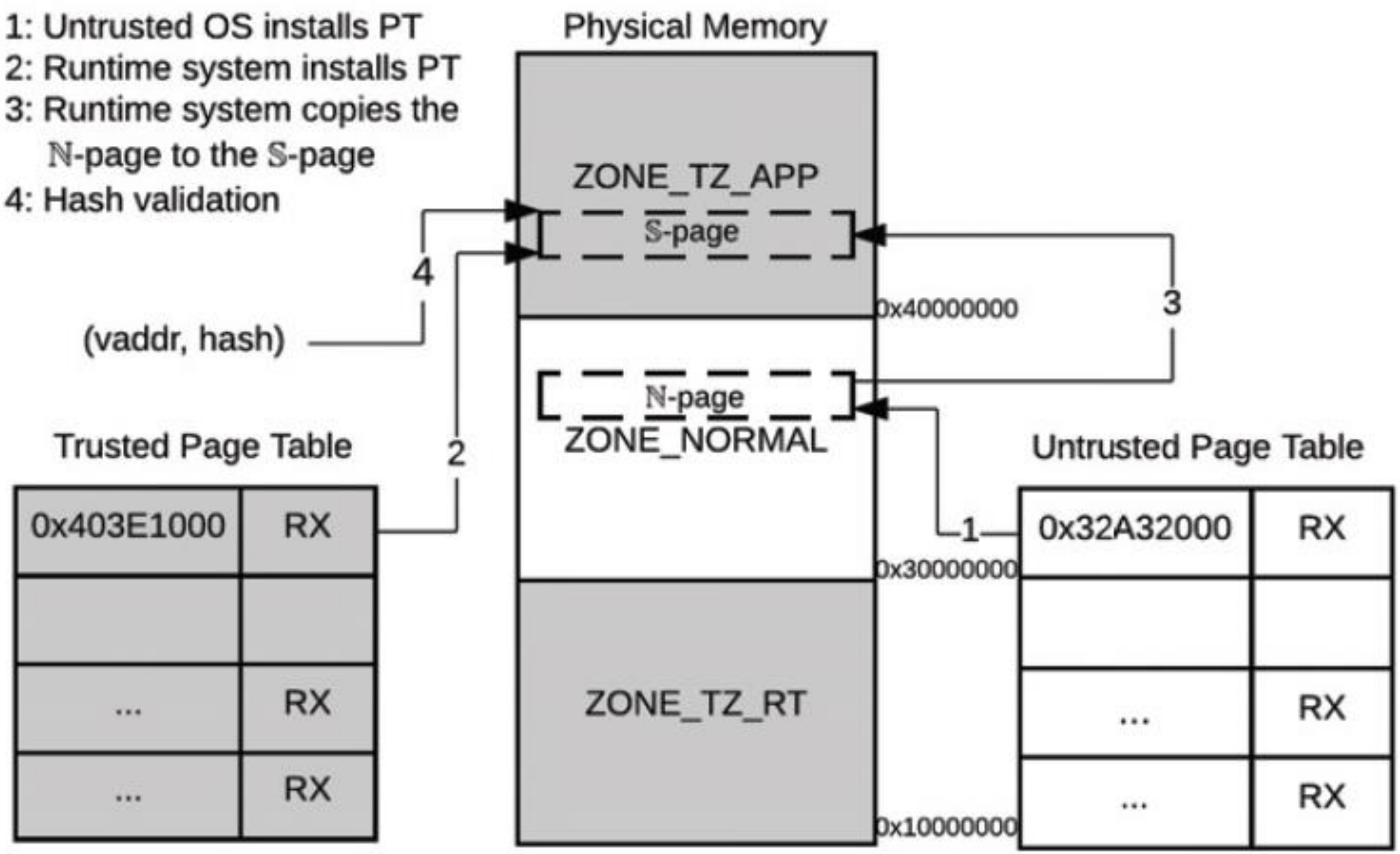


Figure3: PageTableUpdate with integrity check

5) RTS - HANDLING PAGE FAULT

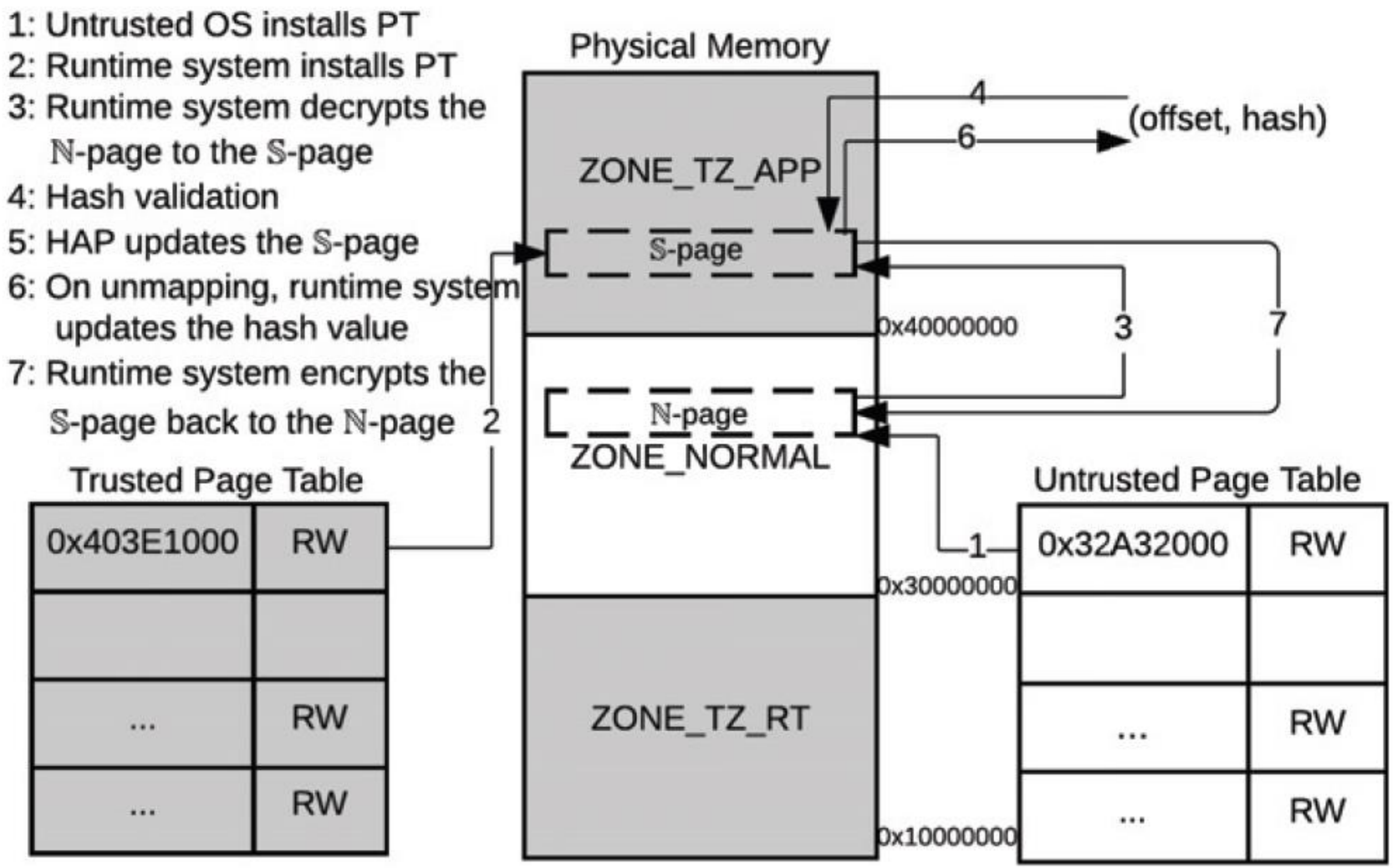


Figure4: PageTableUpdate for Protected Files

5) RTS - INTERVENING SYSTEM CALLS

- OS has no access to user data from shadow HAP
  - system call parameters are values → RTS forwards them directily
  - Pointers: RTS marshals them in a world shared buffer

    →OS gets temporary access to the system call parameters
- procedures for signal handling and coordinating Futex

  - Defeating Iago Attacks
  - Manipulate return of system call → leak used for return oriented programming
  - RTS checks the results for memory overlaps
  - If one is found: → HAP is killed

5) RTS - INTERNAL EXCEPTION HANDLING

Floating Point Computation

- Multiple processes enter VFP – Linux maintains VFP context for each process
  - Leaks User Data
- RTS duplicates code handling VFP

Random Number Generator

- Random numbers very important for cryptographic operations
- OS should not know key materials
- RTS utilizes on-board hardware RNG4

5) RTS - MANIFEST DESIGN

- Each HAP is bundled with a manifest
  - Provides meta data for security features
  - Per application secret key
  - Integrity metadata (vaddr, hash)
  - List of filenames that should be protected
- Manifest is stored on persistent storage
  - Encrypt per-application key by per-device public key
  - Append digital signature

6) IMPLEMENTATION

Normal World – changes on linux

- Added parameter to indicate ZONE_TZ_APP -> pages for HAPs come from this region
- Added a flag -> OS can distinguish HAPs
- New System call to start HAPs
- Changed ret_to_user -> OS pass execution back to shadow instead of zombie
- Hooked page fault handler
- Modifeid code handling signals

→ 300 LOC

6) IMPLEMENTATION

Secure World

→ 4.5 k LOC in C + 0,8k LOC of assembly

- Applicable for manual review or formal verification
- In addition: secure boot mechanism

7) EVALUATION

Microbenchmarks

- Overhead imposed by system calls
- Ran each benchmark with 1,000 iterations -> took average

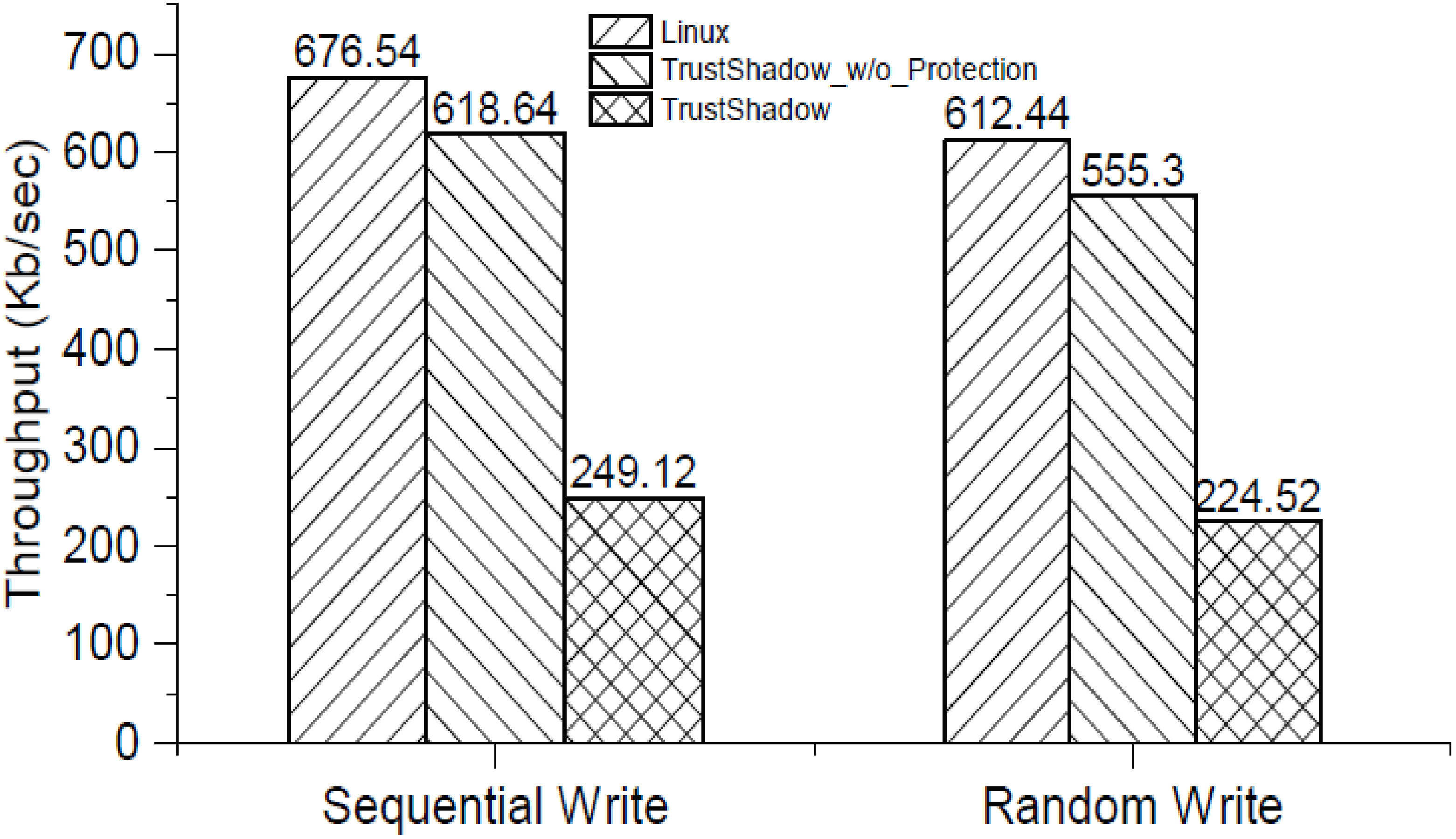
	Latency (μs)		Overhead		
Test case	Linux	Trust Shadow	Trust Shadow	InkTag	Virtual Ghost
null syscall	0.7989	1.6048	2.01x	55.80x	3.90x
open/close	29.2168	40.7886	1.40x	4.83x	7.95x
mmap (64m)	559.0000	784.0000	1.40x	4.70x	9.94x
pagefault	4.7989	7.9764	1.66x	1.15x	7.50x
signal handler install	1.6257	3.8294	2.36x	3.24x	-
signal handler delivery	51.6111	57.0349	1.11x	1.61x	-
fork+exit	987.0000	2328.6000	2.36x	4.40x	5.74x
fork+exec	1060.3333	2509.0000	2.37x	4.20x	3.04x
select (200fd)	15.0707	18.8649	1.25x	3.40x	-
ctxsw 2p/0k	30.3700	32.7100	1.08x	-	1.41x

7) EVALUATION

File Operations

- 128 files, each 8Mb
- Sequential + random write
- Caching disabled

- File protection on → high overhead
- Encryption + hashing
- Solution: better cryptographic engine

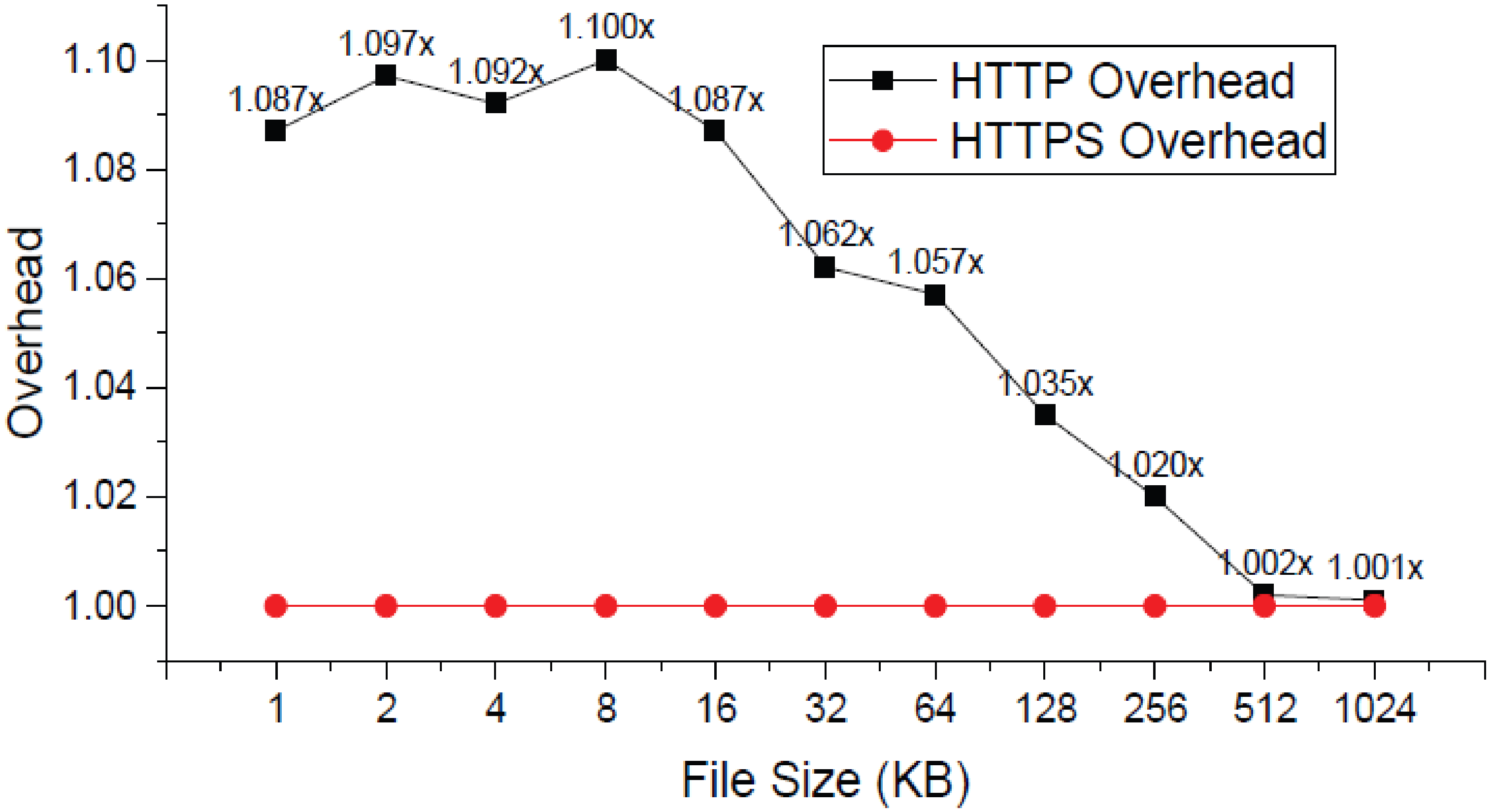


7) EVALUATION

Embedded Web Server

- Impact on real world application
- Respond with HTML files in different size

- *Small files*: reduce troughput ~ 6-10%
- *Big files*: only ~2% from 256 kb
- HTTPS: TS-overhead overwhelmed by intensive cryptographic operations
- Latency: almost no overhead



8) FUTURE WORK

Remaining Attack Surface

- DoS-attacks: process sceduling / start application in normal world
- Manipulation of Manifest
  - Roll-back attack possible
  - Future: version number in manifest
- Side channel attacks still are possible
  - It is possible to adopt known techniques for prevention
  - E.g. cryptographic libraries like OpenSSL
- Physical attacks
  - Solution: store sensitive data on SoC components: harder to compromise
  - Future: extend iRAM

THANK YOU

BACKUP

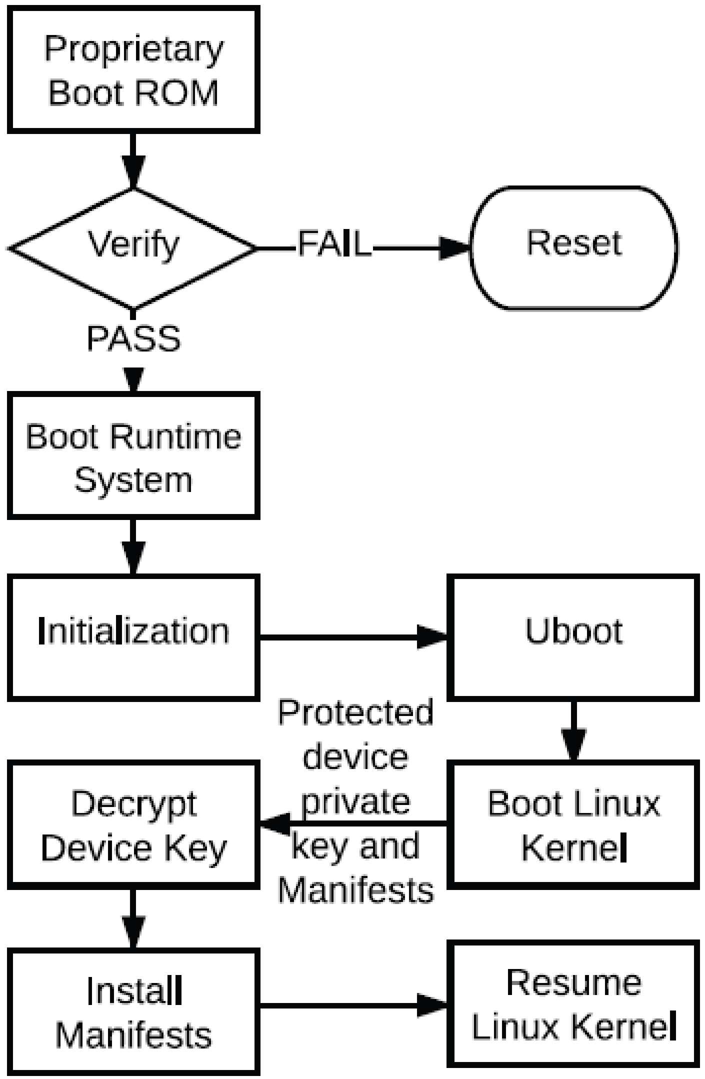
Application level view — System level views — Privileged modes — Exception modes

Application level view	User mode	System mode	Hyp mode †	Supervisor mode	Monitor mode ‡	Abort mode	Undefined mode	IRQ mode	FIQ mode
R0	R0_usr								
R1	R1_usr								
R2	R2_usr								
R3	R3_usr								
R4	R4_usr								
R5	R5_usr								
R6	R6_usr								
R7	R7_usr								
R8	R8_usr								R8_fiq
R9	R9_usr								R9_fiq
R10	R10_usr								R10_fiq
R11	R11_usr								R11_fiq
R12	R12_usr								R12_fiq
SP	SP_usr		SP_hyp†	SP_svc	SP_mon‡	SP_abt	SP_und	SP_irq	SP_fiq
LR	LR_usr			LR_svc	LR_mon‡	LR_abt	LR_und	LR_irq	LR_fiq
PC	PC								
APSR	CPSR								
			SPSR_hyp†	SPSR_svc	SPSR_mon‡	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq
			ELR_hyp†						

† Hyp mode and the associated banked registers are implemented only as part of the Virtualization Extensions

‡ Monitor mode and the associated banked registers are implemented only as part of the Security Extensions

SECURE BOOT


Proprietary Boot ROM
Verify
FAIL
Reset
PASS
Boot Runtime System
Initialization
Uboot
Protected device private key and Manifests
Decrypt Device Key
Boot Linux Kernel
Install Manifests
Resume Linux Kernel