Operating System Support for Run-Time Security with a Trusted Execution Environment

Ph.D Defense
Javier González

March 20, 2015
Innovation

Digital Society

Users

Information Flow

Service Providers

Sensitive!

Personal Devices

- passwords
- ssh-keys
- mails
- certificates
- media content
- apps
- pictures
Digital Society

Users

Innovation

Distrust

Information Flow

Service Providers

Personal Devices

- pictures
- passwords
- media content
- ssh-keys
- mails
- certificates

- apps
Digital Society
Software is guilty

- We have **no control** over how sensitive data is handled in our own personal devices or in the cloud
  - Bad policies, vulnerabilities, weak defaults, data leakage

- It is very difficult to manage the **complexity** of the software enabling innovative services ( compilation- and run-time)
  - Complexity -> Unspecified behaviour -> Indeterminism

- “World-changing innovation is entirely in the code” - Marc Andreessen
Digital Society

Innovation

Trust

Usage Control

Information Flow

Users

Service Providers

Personal Devices
How do we implement support for usage control in commodity Operating Systems in order to protect from today’s cyberattack?

Usage Control is the basis for Run-Time Security
Agenda

1. Run-Time Security
   - Usage Control
   - State of the Art
   - Problem
   - Contributions

2. Split-Enforcement
   - Requirements
   - Design Space
   - State Machine
   - Usage Policy Enforcement

3. Prototype
   - Hardware
   - Trusted Cell
   - Trusted Modules
   - Reference Monitor

Evaluation

Conclusion + Future Work
Usage Control

What is it?

- Enforcement: A priori control of usage rights (access control)
- Audit: A posteriori control of how rights were use
- Usage control is about **who** accesses an object and **how** that object is being used

Usage Control

Attempts to implement it

- Reference monitor (policy engine) resides in the same environment to which the policies apply
- The enforcement of the usage decision occurs also in the same environment
- Examples: Proxos, Taintdroid, Saint Framework,…
Usage Control
How it should be

- Considering a *realistic* attack model:
  - The area where the usage policies apply must be considered *untrusted*
  - … that means that the whole commodity OS stack is *untrusted*!

- As a consequence:
  - The policy engine (reference monitor) must be isolated from the OS
  - The enforcement of usage decisions must also be isolated
Run-Time Security

Overview

- Identify **Sensitive Assets**
  - Any software or hardware component that manages sensitive information: sensitive data, software resources, and peripherals.

- Define a **TEE** (security perimeter) to protect sensitive assets

- Enable **Run-Time Security Primitives** that allow innovative (untrusted) services to make use of these sensitive assets.
  - Innovative services and the commodity OSs are **untrusted**
  - Reference monitor mediates access and usage of sensitive assets

- Provide commodity OSs with **Trusted Services**
Run-Time Security
Overview

Untrusted Area

Trusted Area

Hardware Separation

REE

Applications

Operating System

Memory

TEE

Trusted Components

Reference Monitor

Memory

Secure Hardware

System Bus

Peripherals

Device

Trusted Services

Usage Control

Sensitive Assets

Run-Time Security Primitives

Run-Time Security Primitives

Sensitive Assets

Usage Control

Trusted Services

Run-Time Security Primitives

Sensitive Assets

Usage Control

Trusted Services
Narrowing down
What are you trying to solve?

› Provide support for usage control in commodity OSs
  - Policy engines might vary; we are interested in the framework that can feed any policy engine and enable the generation of usage policies

› Provide support for run-time security in commodity OSs
  - Usage control models could generate a perfect set of policies, but that is not enough if they are not enforced
  - Run-Time security is about enforcing usage control!

› Assume a realistic attack model responding to today’s attacks
  - Both the operating system and innovative applications are untrusted
  - Focus on guaranteeing confidentiality and integrity of sensitive assets

› In terms of the process:
  - We follow an experimental approach - systems research
  - We aim at high adoption and value open-source
Contributions

‣ Exhaustive analysis of the state of the art in run-time security
‣ Hypothesis: Split-Enforcement
  - State machine capturing usage of sensitive assets in UAs
  - Model for a reference monitor to generate usage decisions
  - Two-phase mechanism that allows to enforce usage policies
‣ TEE Support for Linux-based systems
  - Make Open Virtualization to an usable level
  - Generic TrustZone Driver for Linux kernel (being upstreamed)
‣ Trusted Cell
  - Flexible framework to provide trusted services
‣ Trusted Services
  - Reference monitor (enforced usage control) using split-enforcement
  - Trusted Storage solution
  - Certainty Boot (user-centric boot protection mechanism)
Run-Time Security
- Usage Control
- State of the Art
- Problem
- Contributions

Split-Enforcement
- Requirements
- Design Space
- State Machine
- Usage Policy Enforcement

Prototype
- Hardware
- Trusted Cell
- Trusted Modules
- Reference Monitor

Evaluation

Conclusion + Future Work
Requirements

‣ The untrusted area is untrusted (yes, the whole software stack!)

‣ The reference monitor should generate usage policies based on the use of sensitive assets by untrusted area (UA)

‣ These usage policies must be enforced

‣ Run-Time security policies must be available to the UA

‣ The UA must not be constrained in terms of its software stack

‣ The Trusted Area (TA) should minimize complexity (low TCB)

‣ Confidentiality and integrity of sensitive assets is paramount

‣ We must be able to experiment with the solution!
Trusted Services

Design Space

Untrusted Area

- Commodity OS
  - Process (untrusted app)
  - Generic Interface (e.g., VFS)
  - Implementation (e.g., ext4)
  - Device Drivers (e.g., MTD)

Trusted Area

- Secure Area
  - Reference Monitor
  - Secure Components
    - secure VFS
    - secure ext4
    - secure MTD driver

Hardware

- CPU
- Secondary Storage (i.e., NVRAM)
- Memory
- Other peripherals (e.g., Ethernet)

Usage decision enforced in untrusted area
- Usage decision enforced in secure area - use of secure drivers
- Usage decision enforced in secure area - split enforcement
Trusted Services

Design Space

- State of the Art
  - Use of a Trusted Area to generate usage decision (in the best case)
  - OS is trusted - leap of faith
Trusted Services

Design Space

Trusted Area

Secure Area

Reference Monitor

Secure Components

Secure ext4

secure MTD driver

Secure ext4

Usage decision enforced in secure area - use of secure drivers

Usage decision enforced in secure area - split enforcement

Trusted Services

Design Space

Untrusted Area

Commodity OS

Process (untrusted app)

Generic Interface (e.g., VFS)

Implementation (e.g., ext4)

Device Drivers (e.g., MTD)

Usage Driver

Usage decision enforced in untrusted area

Hardware

CPU

Secondary Storage (i.e., NVRAM)

Memory

Other peripherals (e.g., Ethernet)

Secure ext4

Usage decision enforced in secure area - use of secure drivers

Usage decision enforced in secure area - split enforcement
Trusted Services

Design Space

- **State of the Art**
  - Use of a Trusted Area to generate usage decision (in the best case)
  - OS is trusted - leap of faith

- **Secure Drivers**
  - Large Trusted Computing Base (TCB)
  - Replicated complex and community tested functionality - vulnerabilities!
  - Imposed software stack - constrained innovation
  - Reference: seL4, MirageOS
  - Certification != Formal Verification
Trusted Services
Design Space

![Diagram showing trusted and untrusted areas, hardware components, and secure components.]

- **Trusted Area**
  - Secure Area
    - Reference Monitor
    - Secure Components
      - secure VFS
      - secure ext4
      - secure MTD driver
  - Secure Area
    - Usage Driver
      - Usage decision enforced in secure area - use of secure drivers
      - Usage decision enforced in secure area - split enforcement
- **Untrusted Area**
  - Commodity OS
    - Process (untrusted app)
  - Generic Interface (e.g., VFS)
  - Implementation (e.g., ext4)
  - Device Drivers (e.g., MTD)
  - Usage Driver
    - Usage decision enforced in untrusted area

- **Hardware**
  - CPU
  - Secondary Storage (i.e., NVRAM)
  - Memory
  - Other peripherals (e.g., Ethernet)

Legend:
- Red arrows: Usage decision enforced in untrusted area
- Green arrows: Usage decision enforced in secure area - use of secure drivers
- Blue arrows: Usage decision enforced in secure area - split enforcement
Trusted Services

Design Space

- **State of the Art**
  - Use of a Trusted Area to generate usage decision (in the best case)
  - OS is trusted - leap of faith

- **Secure Drivers**
  - Large Trusted Computing Base (TCB)
  - Replicated complex and community tested functionality - vulnerabilities!
  - Imposed software stack - constrained innovation
  - Reference: seL4, MirageOS
  - Certification ≠ Formal Verification

- **Split-Enforcement**
  - Small TCB
  - Locking/Unlocking sensitive assets

Today’s Attack Model
Trusted Services

Design Space

KEEP CALM AND KEEP'EM SEPARATED

Secure Drivers

Split-Enforcement
Split-Enforcement

Formal Model

- Represent usage of sensitive assets
  - State machine abstraction
  - States describe untrusted applications, transitions depend on sensitive asset use
  - Controlled in Trusted Area

- Operation
  - UAs begin and finish in untrusted state
  - Transitions strictly governed by which sensitive asset the UA is using
  - State is used by the reference monitor (RM) as application metadata to generate a usage decision
  - The PDP keeps a historical record of sensitive assets in use for each UA in sensitive state (i.e., mutability)
Split-Enforcement

Challenge

We can generate a usage decision, and represent usage of sensitive assets by the Untrusted Area...

... but, how do we enforce the usage decision?
Locking Mechanisms

Overview

- They enforce usage policies generated by the reference monitor
- They allow to maintain a low TCB in the Trusted Area
- We identify three different types of locking mechanisms:

**Peripheral Locking**
- Mediate usage of sensitive peripherals in the system bus (e.g., Ethernet, I/O devices)

**Data Locking**
- Mediate usage of sensitive data at rest

**Memory Locking**
- Mediate usage of sensitive data in motion
Locking Mechanisms
Peripheral Locking

- **Peripheral setup**
  - Assign peripherals to TA & UA
  - Protected by the hardware
  - Configure at run-time
  - Gatekeeper

- **Unlocking**
  - Mediation: reference monitor
  - Only from Trusted Area

- **Locking**
  - Sensitive per. locked *a priori*
  - Re-lock: Capture interrupts

- **Technology support**
  - Protection in system bus
  - TrustZone is good alternative, but there are others!

Locking Mechanisms

Data Locking

- **Decouple encryption and decryption operations**
  - **Locking**: Encryption
  - **Unlocking**: Decryption
    - Mediation: Reference monitor

![Diagram of Locking Mechanisms]

- Usage decision enforced in untrusted area
- Usage decision enforced in secure area - use of secure drivers
- Usage decision enforced in secure area - split enforcement
Mediate memory accesses in UA through the reference monitor
- All memory accesses are captured (trap-and-emulate)
- Track memory operations to enforce usage policies - lock/unlock
  - We provide the framework; a model to track memory is necessary (e.g., CFI, CPI)
- Example:
  - If sensitive data is given access to the untrusted area (e.g., contact list) a given peripheral cannot be accessed (e.g., Ethernet interface)
Split-Enforcement

What is new?

- Sensitive assets are directly accessible by innovative applications without imposing software stack

- Usage control policies can be enforced while (i) maintaining a minimal TCB, and (ii) not replicating time-tested code

- Confidentiality and integrity of sensitive assets is guaranteed
Split-Enforcement

Hardware Support

- **Secure Peripherals**
  - TrustZone supports on-the-fly assignment of peripherals to the trusted and untrusted areas

- **Memory Partition**
  - TrustZone supports to allocate isolated memory to the trusted and untrusted areas

- **TEE**
  - TrustZone provides an extra privilege mode to support a trusted area (user space - kernel space - secure space)

- **Trusted - Untrusted Integration**
  - Dynamic security perimeter at run-time
  - Cost: no tamper-resistance

TEE Framework
Designing the Trusted Cell

‣ Requirements for the Trusted Area
- Largely independent TEE, high duty cycle, and ~real-time constrains
- Synergy with the untrusted area: Generating and enforcing usage policies
- Open source: Experimental research in academia

‣ Design limitations
- TrustZone has been a closed, obscure technology since 2004
- 3 years ago, most hardware manufacturers disabled the TrustZone security extensions. TTBOOK, only on ARM’s VE and Xilinx’s Zynq*
- Open Virtualization (OV) was the first open-source project leveraging TrustZone (only one back in 2012).
- OV was incomplete, unstable, and targeted Global Platform’s traditional TrustZone use cases (offload of low duty cycled secure tasks)… but it was a place to start!

*Today TrustZone available on: Freescalei.MX53, Samsung S3C6410 SoC, Nvidia Tegra 3 and 4, Qualcomm Snapdragon 400 MSM8630, …
TEE Framework
Setup + Contributions

‣ Commercially available hardware
  - Xilinx Zynq-7000 ZC702
  - Public secure registers (http://www.xilinx.com/support/documentation/user_guides/ug1019-zynq-trustzone.pdf)
  - Uboot, QEMU, Device Tree, compilers, gdb, etc. (https://github.com/xilinx)

‣ TEE
  - Open Virtualization
  - New Features:
    • Communication Interface
    • Updated OpenSSL (HeartBleed)
    • Refactoring (git friendly) + Porting to kernel submodule + Publication + Documentation [1]
  - Bug Fixing:
    • L2 cache
    • Memory management: 2 memory leaks (4KB and 32KB) - non-GP use cases
  - Generic TrustZone driver for Linux kernel (LKLM)

Split-Enforcement Prototype

ZC702

OV: http://www.openvirtualization.org
Linux-xlnx: https://github.com/TrustZoneGenericDriver/linux-xlnx
Ubuntu for Zynq: http://javigon.com/2014/09/02/running-ubuntu-on-zynq-7000/
Trusted Cell
Overall Architecture

Untrusted Area
- Applications
  - REE Trusted Cell
    - Trusted Cell Manager
    - GP Client API
    - TEE API_1
    - TEE API_2
    - TEE API_n
  - Proprietary TEE API
  - LSM Hooks
  - tz_device

User Space
- Libraries (e.g., libc)
- sysfs
- (Secure) syscalls

Kernel Space
- Kernel Subsystems
  - REE Trusted Cell
    - LSM Trusted Cell
  - TrustZone Kernel Driver
    - tz_generic
    - tz1
    - tz2
    - tzn

Trusted Cell Manager
- TIM
  - hash(metadata)
  - hash(data)
  - verify(file)
- TSM
  - encrypt(X)
  - decrypt(X)
- TUM
  - create(policy)
  - modify(policy)
  - evaluate(policy)
- TSPM
  - Run-time Security Primitives
  - dispatch(secure_call)

Secure Area
- TC Logs
- Metadata
- CRYPTO
- Policy Engine (e.g., UCON)
- Service Usage Contract
- Device Usage Policy

Untrusted Area
- Untrusted Device Drivers

Memory
- Peripherals connected to System Bus (e.g., secondary storage)
- TZPC

Trusted Device Drivers
- Tamper Resistant Unit (secure Element)
- Crypto Keys
Trusted Cell

Reference Monitor

- Used Trusted Modules
  - TSPM: Expose “Reference Monitor” services to Untrusted Area
  - TSM and TIM: Cryptographic operations and trusted logs
  - TUM: Usage policy engine
    - Peripheral Locking
    - Memory tracing (reference implementation)

- Functionality
  1. REE LSM feeds TUM with untrusted application’s system calls
  2. TUM generates a usage decision based on policies
  3. Use of sensitive assets are used -> UA sensitive state (mutability)
  4. Sensitive asset and memory usage are mediated (lock/unlock)
    - Peripheral Locking
    - Memory Locking
  5. Sensitive assets are unlocked
  6. Untrusted application returns to untrusted state
Trusted Cell Prototype

- Modules: TIM, TSM, and TSPM fully implemented
  - Encryption, hashing, and verification
  - Dispatcher to trusted services using REE Trusted Cell
  - Working peripheral locking and data locking

- Shortcomings
  - Hardware-Assisted Encryption
    - No NVRAM and no PCI-e in Zynq (prototype done with SD Card)
    - SSDs typically embed a proprietary FTL (but do have cryptoprocessors) - Future work!
  - Memory Locking
    - TrustZone does not support trap-and-emulate in MMU (2 MMUs - secure/non-secure world)
    - Microblaze (FPGA) does not support this either
    - No platforms with Virtualization Extensions and TrustZone enabled (doc, community, …)

➡ Provide proof-of-concept implementation!
  - We explored the design space to its end - Zynq cannot provide more
  - More engineering would not change decisions, or conclusions
Run-Time Security
- Usage Control
- State of the Art
- Problem
- Contributions

Split-Enforcement
- Requirements
- Design Space
- State Machine
- Usage Policy Enforcement

Prototype
- Hardware
- Trusted Cell
- Trusted Modules
- Reference Monitor

Evaluation

Conclusion + Future Work
Analytical Evaluation

Requirement Compliance

- Untrusted Area is untrusted (commodity OS + applications)
  - Split-Enforcement *does not* depend on the untrusted area!
- Split-Enforcement (State Machine and Reference Monitor)
  - Allows to generate and enforce usage policies with an untrusted OS (TC)
- Low TCB
  - TrustZone driver + Trusted Cell ~7K LOC (<10K LOC is in the range for formally verification) - OV (~30K LOC), OpenSSL (~400K LOC)
- Security + Innovation
  - The untrusted area is not constrained (scope, framework, nor resources)
  - No (complex) code replication
  - Trusted Cell exposes trusted services to untrusted applications
  - Confidentiality and integrity of sensitive assets is guaranteed
Experimental Evaluation

Application Benchmarks

- Kernel compilation with full monitoring (∀syscalls) < 18% overhead
Experimental Evaluation
Micro Benchmarks

- Memory allocator in TEE has a huge impact (OV: memory pool)
- Overhead of calls to TEE in < 10µs range (5.3µs in avg.)
Conclusion
Split-Enforcement on TrustZone

- Data locking
  - I/O latencies today measured in 100s of μs. in most advanced datacenters today

- Memory locking
  - Memory operations are measured in ns.; μs. represent a big overhead - trade-off
  - Expect better results with full virtualization

- Peripheral locking
  - Access to APB bus measured in ns.
  - But, …communication with peripherals in ms. or sec. (specially is humans involved)
  - Variable latencies

- In ARMv8 we expect less overhead by implementing time-critical locking (e.g., memory locking) directly in the secure monitor (EL3)
Experimental Evaluation
Split-Enforcement on TrustZone

- The cost of a secure round trip is in the range of the few microseconds (5.3μs in avg. at 666MHz.)
- Allows to generate and enforce usage policies with an untrusted OS
- The overhead of 10000s context switches, executing 1000s of instructions in secure space < 18%
- The Trusted Execution Environment provided by Open Virtualization is not well suited for long-running tasks in secure spaces
  - Secure tasks run to completion without being preempted
  - Pre-allocated memory pool
Agenda

1. Run-Time Security
   - Usage Control
   - State of the Art
   - Problem
   - Contributions

2. Split-Enforcement
   - Requirements
   - Design Space
   - State Machine
   - Usage Policy Enforcement

3. Prototype
   - Hardware
   - Trusted Cell
   - Trusted Modules
   - Reference Monitor

4. Evaluation

5. Conclusion + Future Work
How do we implement support for usage control in commodity Operating Systems in order to protect from today’s cyberattacks?

Run-Time Security

• Split-Enforcement
  - Sensitive assets are directly managed by the untrusted area (low TCB)
  - Locking/Unlocking mechanisms enforce usage policies
  - Untrusted Area can support innovative, complex services

• Operating System Support
  - TEE support for Linux: Open Virtualization + TrustZone driver
  - Trusted Cell: Modular framework to provide different trusted services
  - Prototype:
    • Experimental evaluation in real hardware
    • TrustZone Support + Trusted Cell + Reference Monitor + Trusted Storage + Certainty Boot
Generic TrustZone Driver

Current Status

- First patch-set sent to LKML
- Leading new patch-set
  - Collaborators: Linaro, STMicroelectronics, Huawei, and others
  - Mailing List: (tee-dev@lists.linaro.org)
  - Development: (https://github.com/TrustZoneGenericDriver)
  - Proposed new submodule: /drivers/sec-hw (TrustZone, TPM, …)

### Subsystem Lines of Code Modified
<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Lines of Code Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic TrustZone Driver</td>
<td>+ 519</td>
</tr>
<tr>
<td>Sierraware OTZ driver</td>
<td>+ 1511, - 423</td>
</tr>
<tr>
<td>Total</td>
<td>+ 2030, - 423</td>
</tr>
</tbody>
</table>

(http://lwn.net/Articles/623380/)
(http://www.phoronix.com/scan.php?page=news_item&px=MTg1MDU)

### User Space

- OP-TEE Client (SPECIFIC)
- OV Client (SPECIFIC)
- TEE Client (COMMON?)

### Generic TrustZone Interface (COMMON)

- OP-TEE (SPECIFIC)
- OV (SPECIFIC)
- Others (SPECIFIC)

### TrustZone Supported Chips (COMMON)

- Target 1
- Target 2
- Target 3
- Target n

### smc call

- Open/Close (Posix)
- TEE_IOC_CMD
- TEE_IOC_TASK
- TEE_IOC_SHM_ALLOC
- TEE_IOC_SHM_FREE
- TEE_IOC_LOAD_TS
- TEE_IOC_GET_TS_LIST
- …

### IOCTL (ongoing discussion)

- Ongoing discussion
Future Work

› **Current Trusted Services**

- Trusted Storage
  - I/O Trusted Cell
  - Trusted Cell Communication
  - Trusted Area DMA

- Reference Monitor
  - Port to ARMv8: TrustZone + Virtualization Extensions
  - Use a model to trace memory operations (e.g., CFI, CPI)

› **New Trusted Services**

- Trusted Service Installer
- Trusted Service Store
- Trusted Area Intrusion Detection (using TIM)
- Trusted Cell Verification (using TIM and Certainty Boot)
- Split-Enforcement in other Secure Hardware (e.g., Intel SGX)
Operating System Support for Run-Time Security with a Trusted Execution Environment

Ph.D Defense
Javier González (javier@javigon.com)

March 20, 2015
Support Slides

Ph.D Defense
Javier González (javier@javigon.com)

March 20, 2015
Trusted Cell
Overall Architecture

Untrusted Area

- REE Trusted Cell
- Trusted Cell Manager
- GP Client API
- TEE API
- Proprietary TEE API
- Applications
- (Secure) syscalls
- Libraries (e.g., libc)
- sysfs
- tz_device

Kernel Space

- REE Trusted Cell
- REE Trusted Cell
- LSM Hooks
- LSM Trusted Cell
- TrustZone Kernel Driver
- tz_generic
- tz1
- tz2
- tzn

User Space

- Libraries (e.g., libc)
- sysfs
- tz_device

Secure Area

- TEE Trusted Cell
- TIM
  - hash(metadata)
  - hash(data)
  - verify(file)
- TSM
  - encrypt(X)
  - decrypt(X)
  - TSM cache
- TUM
  - create(policy)
  - modify(policy)
  - evaluate(policy)
  - TUM queue
- TSPM
  - Run-time Security Primitives
  - dispatch(secure_call)
  - TSM cache

Trusted Cell Manager

- TIM cache
- Policy Engine (e.g., UCON)
- Service Usage Contract
- Device Usage Policy
- TC Logs
- Metadata
- CRYPTO

Policy Engine

- Policy Engine
- Service Usage Contract
- Device Usage Policy

Trusted Device Drivers

- TC driv.
- TC
- driv

Peripherals connected to System Bus
(e.g., secondary storage)

Memory

- TZPC

Tamper Resistant Unit (secure Element)
- Crypto Keys
Trusted Cell
REE Trusted Cell

- **Kernel submodule support**
  - Need for well-known interface - kernel is static at run-time
  - Linux Security Module (LSM) is a perfect match
    - LSM mediates access to kernel space objects in the kernel
    - Set of transversal hooks to all kernel submodules: I/O stack, network stack, etc.
    - Already implements the concept of policies (though for access control only)
    - Inconvenient: Only one LSM at a time*

- **LSM Trusted Cell**
  - New LSM module focused on usage control (AppArmor, SELinux, etc.: for access control)
  - Monitor (classes of) system calls - evaluation with worst case

- **Introduction of security-critical system primitives**
  - Validate integrity of kernel image (certainty boot)
  - Validate boot sequence at run-time (certainty boot)
  - Others: Crypto API, capability subsystem, and other LSM frameworks

*Casey Schauffer has proposed a new LSM stacking mechanism that is having backup from the kernel community. (LKLM)*
Trusted Cell

REE Trusted Cell

- User space application support
  - Less constrained than kernel - user space is dynamic at run-time
  - Different APIs can be implemented on top of \textit{tz\_device}
    - We implement a subset of Global Platform's Client API - proof of concept
    - Support TEE-specific (maybe proprietary) libraries
  - TEE-frameworks can define their \textit{iocltls}
    - Support TEE-specific (maybe proprietary) operations (e.g., OP-TEE tee-suppplicant)
  - Possibility to load TEE-specific LKMs directly using kernel interface
    - Support TEE-specific (maybe proprietary) operations

- REE Trusted Cell management through \textit{sysfs}
Trusted Cell
Overall Architecture

Untrusted Area

- Applications
  - REE Trusted Cell Manager
  - GP Client API
  - TEE API

User Space
- (Secure) syscalls
- LSM hooks
- tz_device

User Libraries (e.g., libc)

Sysfs

Kernel Subsystems
- REE Trusted Cell
- LSM Trusted Cell
- TrustZone Kernel Driver
  - tz_generic
  - tz1, tz2, tzn

Untrusted Device Drivers

Secure Area

TEE Trusted Cell

- TIM
  - hash(metadata)
  - hash(data)
  - verify(file)

- TSM
  - encrypt(X)
  - decrypt(X)

- TUM
  - create(policy)
  - modify(policy)
  - evaluate(policy)

- TSPM
  - Run-time Security Primitives
  - dispatch(secure_call)

Trusted Cell Manager

- TIM cache
- TSM cache
- TUM queue

TEE Logs
- TC
- Meta
- Crypto
- Policy Engine (e.g., UCON)
- Service Usage Contract
- Device Usage Policy

Trusted Device Drivers

Memory

Peripherals connected to System Bus
(e.g., secondary storage)

TZPC

Tamper Resistant Unit (secure Element)
Crypto Keys
Certainty Boot
Architecture

Untrusted Area
- Applications
- Mobile OS
- OSBL

Secure Area
- Secure Tasks
- TEE OS
- TEEBL

Secure Element
- Two-phase boot applet
  - Trusted public keys
  - Hash of boot log

Manufacturer
- Second Stage Bootloader (SSBL)
- First Stage Bootloader (FSBL)

Root of Trust - Components
- SE to application processor communication via APB

Verified Components
Certainty Boot

Boot Sequence

(1) Power on

FSBL/SSBL running

Manufacturer

(2) Verify OS/OSBL

[OS/OSBL not verified]

[OS/OSBL verified]

Secure environment & SE

(4) Display unverified components, ask for SS

(5) SE adds issuer public key to list

[correct SS]

[wrong SS or canceled]

Rich environment

(3) System running with SE access

(6) System running without SE access

Desired path

Uncertified result

Exchange OS/OSBL
Hardware
Zynq ZC702
Hardware

TrustZone

- TrustZone-enabled SoC
- Peripherals
- Trusted Area
- Untrusted Area
- Secure World
  - Secure Monitor
  - Secure OS
  - Secure Tasks
- Non-Secure World
  - Applications
- Memory
- Secure World Memory
- Non-Secure World Memory
- Cortex-A / ARM1176 Processor with ARM TrustZone Technology
- Embedded OS
- Normal World
  - Open Applications
  - Applications Requiring Secure OS Support
- Secure OS
- Monitor
- TrustZone API
- TZ Driver
- Secure Element (SecurCore)
- Applications
Hardware

TPM

Untrusted Area
- Communication Bus
- Main Memory
- Peripherals

Trusted Area
- I/O
  - Gate Keeper
- Crypto Engine
- Random Number Generator
- PCR Registers (≥16 registers)
- Non-Volatile Memory (≥ 1280 bytes)
- Volatile Memory
- Execution Engine
- ...

I/O

Non-Volatile Memory (≥ 1280 bytes)

Volatile Memory

Execution Engine

PCR Registers (≥16 registers)
Hardware
Intel SGX

- Untrusted Area
  - Application Code
  - Application Stack
  - Enclave
  - Operating System

- Trusted Area
  - Enclave Code
  - Enclave Stack
  - Enclave Heap
  - Entry Table

Hardware
Peripherals
Hardware
Smart Card

Untrusted Area

Communication Bus
Main Memory
Peripherals

Trusted Area

Interfaces
Smart Card Controller
RAM
FLASH
Software is guilty

- “Software is eating the world” - Marc Andreessen
- “Almost all modern systems share a common Achille’s heel in the form of software” - Gary McGraw
  - The Trinity of Trouble: Complexity, Connectivity, and Extensibility
  - 5 to 50 bugs per 1 KLOC
  - Linux Kernel +15 million LOC today (not all in kernel image)
- **Complexity -> Unspecified Behaviour -> Indeterminism**
  - Intentionally introduced vulnerabilities can be hidden (e.g., back doors)
  - Chances for unintentionally introduced indeterminism augment
  - Corner Stone for software attacks
  - Examples: Heartbleed, ShellShock, Freak

How do we handle the security of complex software without killing innovation?
“World-changing is entirely in the code”
Marc Andreessen

“Almost all modern systems share a common Achille’s heel in the form of software”
Gary McGraw
Example usage policies:

- If sensitive data is given access to the untrusted area (e.g., contact list) a given peripheral cannot be accessed (e.g., Ethernet interface)

- If access to a number of sensitive assets (data, peripherals) has previously been granted, a given set sensitive asset cannot be accessed. Decisions made on previous usage (mutability)
Trusted Cell
TEE Trusted Cell

- Trusted Security Module (TSM)
  - Cryptographic operations (AES-128, SHA-3)
  - OpenSSL (openssl-1.0.1j - October, 2014)
- Trusted Integrity Module (TIM)
  - Integrity verification (e.g., IMA in kernel)
  - Merkle Hash-Tree
  - Trusted logs
- Trusted Usage Module (TUM)
  - Policy engine (Device usage policy)
- Trusted System Primitive Module (TSPM)
  - Run-time security primitives in the that the TEE Trusted Cell
  - Dispatcher to Trusted Modules (Services)


Microsoft’s NGSCB Overview

‣ Motivation
- DRM, vendor lock-in, exclude competitors (e.g., Open Office)
- Bruce Schneier: “…will lead us down a road where our computers are no longer our computers, but are instead owned by a variety of factions and companies all looking for a piece of our wallet.”

‣ Concept
- Full stack solution
- Traditional offload use cases (available)

‣ The technology has not fully materialized
- Motivated BitLocker full disk encryption
- Measured Boot feature in Windows 8 (TPM)
- Subject to timing attacks (by design)
Analytical Evaluation

Security Analysis

Hack Attacks (software)

Shack Attacks (software + simple hardware)

Lab Attacks (complex hardware)

Fab Attacks (design, hardware, firmware)

Guaranteed

Confidentiality and integrity of sensitive assets

Durability and availability of sensitive assets

Out of Scope
Sensitive data never leaves the trusted area unencrypted
- Extends the size limitation of tamper-resistant storage (SE)
- Secure Containers are as resistant as the encryption algorithm
- Provide integrity and confidentiality, not availability or durability*

*Redundancy can help to increase the level of availability and durability, but cannot guarantee it (untrusted area is "fully" untrusted)
Trusted Services

Trusted Storage: Untrusted Data Generation

Untrusted Area

- Commodity OS
- User Space
- Kernel Space
- Memory Operations
- I/O Stack
- Process (untrusted app)

Trusted Area

- Secure Area
- Reference Monitor
  - TIM
  - TSM
  - TUM

Secure Element

- Hypervisor
- TEE Trusted Cell
- Flash
- SDS I/O device
- key management
TrustZone Support
TEE + OS Support

Untrusted Area
- REE (non-secure world)
  - Applications
  - Commodity OS
- TrustZone Driver

Trusted Area
- TEE (secure world)
  - Secure Tasks
  - Secure OS
  - Secure Monitor

Memory
- Non-Secure World Memory
- Secure World Memory

TrustZone-enabled SoC

Peripherals
TEE + OS Support

Untrusted Area
- REE (non-secure world)
  - Applications
  - Commodity OS
  - TrustZone Driver
- Non-Secure World Memory

Trusted Area
- TEE (secure world)
  - Secure Tasks
  - Secure OS
  - Secure Monitor
- Secure World Memory

TrustZone-enabled SoC

Peripherals
TEE Framework
Designing the Trusted Area

- Requirements for the Trusted Area
  - Largely independent TEE, high duty cycle, and ~real-time constrains
  - Synergy with the untrusted area: Generating and enforcing usage policies (TOPPERS SafeG)
  - Open source: Experimental research in academia

- Design limitations
  - TrustZone has been a closed, obscure technology since 2004
  - 3 years ago, most hardware manufacturers disabled the TrustZone security extensions. TTBOOK, only on ARM’s VE and Xilinx’s Zynq*
  - Open Virtualization (OV) was the first open-source project leveraging TrustZone (only one back in 2012).
  - OV was incomplete, unstable, and targeted Global Platform’s traditional TrustZone use cases (offload of low duty cycled secure tasks)… but it was a place to start!

*Today TrustZone available on: Freescalei.MX53, Samsung S3C6410 SoC, Nvidia Tegra 3 and 4, Qualcomm Snapdragon 400 MSM8630, …
TEE Framework
Setup + Contributions

‣ Commercially available hardware
  - Xilinx Zynq-7000 ZC702
  - Uboot, QEMU, Device Tree, compilers, gdb, etc. ([https://github.com/xilinx](https://github.com/xilinx))

‣ TEE
  - Open Virtualization
  - Trigger IP revision
  - New Features:
    • Communication Interface
    • Updated OpenSSL (HeartBleed)
    • Refactoring (git friendly) + Porting to kernel submodule + Publication + Documentation [1]
  - Bug Fixing:
    • L2 cache
    • Memory management: 2 memory leaks (4KB and 32KB) - non-GP use cases

TEE + OS Support

Untrusted Area
- REE (non-secure world)
  - Applications
- Commodity OS

Trusted Area
- TEE (secure world)
  - Secure Tasks
  - Secure OS
  - Secure Monitor

Memory
- Non-Secure World Memory
- Secure World Memory

TrustZone-enabled SoC

Peripherals
Generic TrustZone Driver

Background

- **State of the art**
  - Kernel TEE framework implements its own TrustZone driver (LKM)
  - Necessary to patch the kernel to enable TEE communication (SMC)
  - Mostly focus on Global Platform use cases (low-duty cycle offload)
    - With the exception of TOPPERS SafeG and (somehow) Genode

- **Consequences**
  - Static assignment of secure devices (peripherals, timers, etc.)
  - Required to recompile the kernel when making changes
  - Required to maintain a set of patches (TEE-enabling) on top of mainline kernel
  - Kernel submodules are excluded
  - TrustZone is an underused technology (TrustZone has been a closed, obscure technology since 2004)
Generic TrustZone Driver Proposal

- Support for user space applications (as today)
  - Kernel support for SMC call - PL1 (ARMv7) and EL1 (ARMv8)
  - Support for user space libraries and APIs (e.g., Global Platform)
  - Support different frameworks in one single driver (generic + specific)
  - Minimize overhead (innovative applications + trusted services)

- Support for kernel submodules
  - Allow them to access trusted services:
    - LKMs do not cover this use case (unknown interface to kernel)
    - A generic interface known to the kernel is necessary - upstream
    - A way to describe supported trusted services is necessary (run-time security primitives)
  - Good candidates: IMA/EVM, kernel keyring, and Crypto API
    - IMA/EVM and kernel keyring already use TPM when available
    - Maintainers have shown interest on a generic TrustZone interface
Generic TrustZone Driver
Architecture

Untrusted Area

- Kernel Subsystems
  - TrustZone Kernel Driver
    - Generic Interface (open/close read/write)
      - tz1
      - tz2
      - tzn

Secure Area

- TEE Framework
  - Secure Tasks
  - Task Dispatcher
  - Secure Monitor
  - Secure Kernel

User Space

- Global Platform Client API
- TEE API 1
- TEE API 2
- ... TEE API n
- Proprietary TEE API
- tz_device

Kernel Space

- Applications

Boot-Time

- Device Tree (untrusted area)
- Device Tree (trusted area)
Generic TrustZone Driver
Implementation Challenges

- Give peripherals the correct base address
  - Kernel maps I/O memory for all devices on initialization
  - I/O memory is typically reclaimed on graceful shutdown
  - Kernel tries to map I/O memory for secure devices -> TZPC blocks
  - **Consequence**: TEE frameworks require TEE-enabling patch

➡ Proposal: “Secure Device Tree”
- Extend device tree with “arm-world” tags
- Assign base address to physical address for secure devices
- Decouple device initialization and termination routines from specific system events (boot and halt processes)
- Secure devices are loaded and unloaded on-the-fly
- **No TEE-enabling patch is necessary!**
Generic TrustZone Driver
Implementation Challenges

- Introduce secure read/write
  - Kernel must forward I/O requests to secure devices
  - The TEE serves I/O requests - TZPC blocks untrusted area
  - I/O request forwarding is platform specific - depends on base address
  - Consequence: TEE frameworks require TEE-enabling patch (again)

Proposal: “Secure Device Tree”
- Use base address to forward I/O requests to TEE
- Add TEE node to the Device Tree
- Allow TEE to add information TEE node at boot-time
- One single Device Tree with secure entries: nodes and tags.

Reference implementation:
- Xilinx Linux-xlns TRD 14.5 (kernel v.3.8)
  - [https://github.com/TrustZoneGenericDriver/linux-xlnx (FIXME)](https://github.com/TrustZoneGenericDriver/linux-xlnx (FIXME))
Generic TrustZone Driver
Implementation Challenges

- Port Open Virtualization
  - Port LKM to kernel submodule
  - Adapt to generic interface: decouple responsibilities
  - Major refactoring: support user space (ioctl) and kernel submodules
  - Identify and fix memory leaks
  - Use of "arm-world" to tag secure devices
  - Use of secure read/secure write to forward I/O requests to TEE
  - Port subset of Global Platform’s Internal API - proof of concept
  - Enabled for Xilinx linux-xlnx TRD 14.5
  - Submitted to LKLM - triggered collaboration with Linaro (OP-TEE group)
  - TrustZone gaining traction in Linux community (e.g., Google, Linaro)

(http://lwn.net/Articles/623380/)
(http://www.phoronix.com/scan.php?page=news_item&px=MTg1MDU)
Generic TrustZone Driver

Current Status

- First patch-set sent to LKML
- Working on new patch-set
  - Collaborators: Linaro, STMicroelectronics, Huawei, and others
  - Mailing List: (tee-dev@lists.linaro.org)
  - Development: (https://github.com/TrustZoneGenericDriver)
  - Proposed new submodule: /drivers/sec-hw (TrustZone, TPM, …)
  - Ongoing discussions for secure Device Tree, secure read/write, etc.

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Lines of Code Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic TrustZone Driver</td>
<td>+ 519, - 423</td>
</tr>
<tr>
<td>Sierraware OTZ driver</td>
<td>+ 1511, - 423</td>
</tr>
<tr>
<td>Total</td>
<td>+ 2030, - 423</td>
</tr>
</tbody>
</table>

(kernel interface (sub-modules)

IOCTL (ongoing discussion)
- Open/Close (Posix)
- TEE_IOC_CMD
- TEE_IOC_TASK
- TEE_IOC_SHM_ALLOC
- TEE_IOC_SHM_FREE
- TEE_IOC_LOAD_TS
- TEE_IOC_GET_TS_LIST
- …

User Space
- OP-TEE Client (SPECIFIC)
- OV Client (SPECIFIC)
- TEE Client (COMMON?)

Generic TrustZone Interface (COMMON)
- OP-TEE (SPECIFIC)
- OV (SPECIFIC)
- Others (SPECIFIC)

smc call
- Target 1
- Target 2
- Target 3
- Target n

TrustZone Supported Chips (COMMON)
Experimental Results
Experimental Evaluation

Micro Benchmarks

- REE performs better than TEE, but TEE is not preempted
Experimental Evaluation
Micro Benchmarks

- TEE has always priority over CPU time over REE
Experimental Evaluation
Micro Benchmarks

<table>
<thead>
<tr>
<th>Overhead TrustZone / Kernel %</th>
<th>1KB</th>
<th>100KB</th>
<th>500KB</th>
<th>1MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES CBC</td>
<td>-3.13%</td>
<td>5.15%</td>
<td>7.99%</td>
<td>-5.22%</td>
</tr>
<tr>
<td>AES CRT</td>
<td>-3.37%</td>
<td>5.11%</td>
<td>8.13%</td>
<td>-5.36%</td>
</tr>
<tr>
<td>AES ECB</td>
<td>2.17%</td>
<td>10.51%</td>
<td>13.82%</td>
<td>-1.40%</td>
</tr>
</tbody>
</table>

- CPU-bound + memory ops. confirm < 18% overhead in TEE