EASING ACCESS TO ARM
TRUSTZONE – OP-TEE AND
RASPBERRY PI 3
09/26/16
OP-TEE on Raspberry Pi 3

1. Motivation
2. Anatomy of the Port
3. Kit Availability
4. Device security – Core concepts
5. TrustZone – “what it means”
Motivation

— Next-gen devices need foundational security
— TrustZone and TEEs fast becoming standards for chip level security
— Lack of widespread knowledge among maker community of TrustZone and TEEs
  — Only hardware and software pros can make security happen
  — Requires deep understanding of hardware and OS fundamentals

— Problem: How to make security accessible?
Motivation

- RPi3: Economical board, $35
- Good support (BSP, online fora)
- Quad Core Cortex-A53 V8a
- Poked around startup code, looked like “some” TrustZone features might be present
- Searched for JTAG, found multiplexor setup
- Appeared as if OP-TEE port might be feasible

Presented by Sequitur Labs Inc. 2016
Anatomy of the Port

- Work done almost immediately following flurry of AARCH64 Raspberry Pi fora posts
- 64-bit (AARCH64) kernel
- U-boot
  - Running NS, wrong mode
  - Boot “stubs”, setting cores to EL2, spin loop
- 64 bit runtime (Linaro/Ubuntu)

- details in separate ppt titled “Anatomy of Port”
Anatomy of the Port

--- Challenges
- Effectively no data sheet; full hardware capability a bit of a mystery
- Dive into source code; read stuff available online

--- Approach (details in appendix)
- Modified boot stubs
- Modified u-boot
- Ported ARM Trusted Firmware (ATF) BL31
- Modified memory maps
- Enabled startup debug; JTAG
- Ported OP-TEE
TrustZone on RPi 3: Details

- Raspberry Pi 3 implements Cortex-A/V8a exceptions

**But**
- It does not implement the crypto acceleration instructions
- Linux Device Tree Source (DTS) showed no indication of any security hardware IP
- No TZPC, TZASC, GIC or other proprietary bus/fabric security control interfaces
- No securable memory
- IE., A53 core security state signals not propagated throughout the fabric.

**Conclusion**
- not-securable TrustZone implementation but **GREAT** for education, learning, Trusted Application development and debug.
Kit Availability

- Linaro repo build: Joakim Bech (busybox stack)
  - tftp download/boot
  - fat load
  - All AARCH64

- Sequitur Labs “GlobalPlatform TEE hackathon setup”
  - AARCH64 Linux
  - Raspbian/32 NW runtime
  - Virtualbox Ubuntu/Mint 17 Linux setup (includes all source)
  - RPi3 + Bus-blaster + Debug cable + uSD card (under $100)
Kit Focus

Linux stack:
- Pi3 in widespread prototype use
- Great for development
- Not a mobile device (phone)

OP-TEE:
- Focus is bigger than mobile, it’s the other 90% of the market
- Applications where trust is required apart from payment and DRM
NEXT-GENERATION EMBEDDED SECURITY: KEY CONCEPTS
Problems of a Big Flat Space

- Security must be designed in
- Isolation enables protection
  - Create hard boundaries around assets that must be protected
  - At least two distinct domains: **secure** and **non-secure**
  - Resources
  - Functions
  - Without requiring an external “security chip”
- Management plane
  - Enforcement controls
  - Boundary protections

Presented by Sequitur Labs Inc. 2016
TrustZone is a Strategy

- Cortex-A core security state is orthogonal to processor mode (USR, SVC)
- Two security domains (states): secure, and non-secure
- Physical security domains incorporate:
  - Memory: on and off chip RAM
  - Bus masters and slaves (IP) {SPI, I2C, QSPI, UART ...}
  - Core resources: MMU, cache, interrupts
- Core security state aligned with security domain

- Much more than crypto …
- Programmable (security features):
  - Provisioning
  - Secure boot*, secure updates
  - Secure peripherals (sensors, actuators)*
  - Cryptography (transaction or session) – eg., SSL or ..
  - Trusted controls

* Implementation defined
Enable Critical Foundational Security Features

“Must have” security features for connected devices:

- Separation, isolation and hardware roots of trust
- Secure boot
- Secure firmware updates
- Authentication (encrypted and authenticated boot)
- Key storage and management (including revocation), unlimited key storage (not a fixed number of “slots”).
- Secure data storage

What critical functions and data must be protected on your device?

What are the trust relationships that you care about in your system?

Solutions must:

- Address immediate security issues
- Address future needs

Can the device be managed in a secure manner?

Can cryptographic algorithms be updated?

Can I enable new services in the future?

Presented by Sequitur Labs Inc. 2016
CORTEX-A SECURITY FUNDAMENTALS REVIEW

Core state and signal propagation
Secure v Not-Secure, Monitor
Interrupt Routing
ARM® TrustZone®

On-chip security enclave that provides hardware isolation and protection for sensitive material such as cryptographic keys, algorithms and data

Same technology that secures

- A billion+ devices
  - Samsung
  - BlackBerry
  - Nokia
  - Microsoft

- Payment and bank terminals
- Set top boxes
- ++

Terminology

- TrustZone is hardware
- Trusted Execution Environment (TEE) is software
- Normal World / Rich OS / REE
- Secure World / Secure OS / TEE
- Secure Monitor
- Trusted Application (TA)
- Trusted Connector APIs
- Trusted Application Manager (TAM)
Cortex-A Review: Core State and Signal Propagation, Interrupts

- ARM core propagates security state to bus fabric (matrix32, matrix64)
- Bus masters, slaves and RAM **must** enforce master/slave security state relationship
  - Non-secure master cannot access secure slave, e.g., Linux (non-secure) **cannot** access AES or cryptographic keys
- TrustZone/Secure World: “like a 2nd virtual, secure CPU”
  - Banked registers + MMU (sctlr, ttbr0/ttbr1 …); i.e., MMU, page tables, physical and virtual addresses unique to secure and non-secure worlds respectively.
- 2 Interrupt lines: FIQ (fast interrupt) and IRQ (interrupt).
  - By convention, FIQ is “owned” by secure world
Cortex-A Review: Security Configuration

- ARM core resets into secure state
  - Preparation of secure and non-secure peripherals done by boot firmware prior to “non secure handoff”.

- SCR: secure configuration register
  - security state, interrupt routing (ie., FIQ forcing).
  - NS = 0 → “secure”
  - NS = 1 → “NOT secure”
  - FIQ and IRQ, delivered to current world (== 0) or forced to monitor (==1) and not maskable from that (NS-state) world.
  - Interrupts (FIQ and IRQ) are routed to VBAR-NS, VBAR-S or MVBAR (non-secure, secure and monitor vector base address register handlers) according to SCR.{F,I,NS} bits.

- SCR is not accessible from normal world (does not exist)
Cortex-A Review:
Monitor Mode (or EL3)

- Implicitly secure regardless of SCR.NS
- Execution/address space always secure, i.e., when in monitor mode and NS=1 and MMU enabled, address translation is via Secure ttbr0/ttbr1.
- But, as above, read/write ttbr0/ttbr1 accesses NON-secure ttbr0/ttbr1 instances.
- i.e., always secure, NS selects Secure or Non-Secure banked registers.

- Monitor mode code, Keep It Simple (KISS): “pass through between worlds” and “context switch”.

Presented by Sequitur Labs Inc. 2016
GETTING STARTED WITH OP-TEE

Planning and Requirements Definition
Writing Trusted Applications
OP-TEE APIs

OP-TEE implements Global Platform compliant APIs:

**TEE Internal Core API**
- The Internal API is the API that is exposed to the TAs running in the secure world. The TEE Internal API consists of four major parts:
  - Trusted Storage API for Data and Keys
  - Cryptographic Operations API
  - Time API
  - Arithmetical API

**TEE Client API**
- Communications API for connecting *Client Applications* running in a rich operating environment with TAs running inside OP-TEE (ie Linux)
TEE Client API

The mechanism to interact with Trusted Applications in the TEE

Standardized
- Compliance tested

Application independent
- Say how messages are passed, not what they mean

Implementation independent
- The same regardless of how the TEE is implemented
- The same regardless of the communications mechanism to the TEE

Efficient
- Capable of zero copy operation if TEE uses correct interface

Blocking
- Simple, easy to understand, programming model for modern operating systems

Easy to use
TEE Client API: Error Handling

Most routines return a return code of type **TEEC_RESULT**

- Many standard values
  - **TEEC_SUCCESS** = 0 always means success

Routines which communicate with the TEE/TA also return error origin indicating where error occurred

- **TEEC_ORIGIN_API**
  - TEE Client API implementation
- **TEEC_ORIGIN_COMMS**
  - Communications between REE and TEE
- **TEEC_ORIGIN_TEE**
  - Shared TEE code outside the TA
- **TEEC_ORIGIN_TRUSTED_APP**
  - Trusted Application defined
TEE Client API:
Key Concepts

— Context:
  Connection from REE to OP-TEE.

— Session:
  Connection from REE app to OP-TEE TA.

— Command:
  Unit of Communication from REE app to TA.

— Shared Memory:
  Shared buffers allocated by Client API or by REE app and managed by OP-TEE driver
Connection to a TEE is represented as a `TEEC_Context`
- Connect using `TEEC_InitializeContext`
- Disconnect using `TEEC_FinalizeContext`
- Always use NULL as the TEE name
Can connect to one or more Trusted Applications in a TEE
   - Identified by a TEEC_Context

Represented by a TEEC_Session object
   - Connect using TEEC_OpenSession
   - Disconnect using TEEC_CloseSession

TA is identified by a UUID

- Specifies the login method and associated data
- Can supply an operation and parameters
- Can be cancellable
Commands identified by a 32-bit integer

Trusted Applications define the meaning of all commands

- One command means different things to different TAs
- Client API does not limit command meaning in any way

Commands are always blocking

- Optional support for cancellation via multi-threading

Commands are sent to the TA along with up to four parameters

- Parameters are combined into a single structure: an Operation
- Parameters can be of three types – **IN, INOUT, OUT**
- All parameters are optional
- TEEC_InvokeCommand
Debug

— Trusted Application/secure world:
  — JTAG, only if “secure JTAG” is enabled
  — DMSG/IMSG/EMSG, ~printf

— TEE Client (Linux):
  — Linux native code; via object or library (.o, .a, .so)
  — gdb
  — printf
Naming: Universally Unique ID (UUID)

— A UUID is effectively unique if randomly generated
— 128 bits
— Normally written in a standard format:
  — 123e4567-e89b-12d3-a456-426655440000
— Every Trusted Application is identified by a UUID
— UUIDs are defined by RFC 4122

— Five formats, only two are important
  — Random UUID has 122 bits of randomness
  — Derived from a SHA-1 hash value
— Impersonation is possible
  — Nothing to stop two Trusted Applications using the same UUID
  — Provable UUID ownership scheme in Administration Framework (see later)
Operation and Parameter Types

Represented as a C structure:

```c
typedef struct
{
    uint32_t started;
    uint32_t paramTypes;
    TEEC_Parameter params[4];
    <Implementation-Defined Type>
    imp;
} TEEC_Operation;
```

`paramTypes` is a combination of the types of the four parameters

- Built using `TEEC_PARAMS_TYPE` macro

Parameter Types

- **IN**: Values passed to the TA
- **OUT**: Values Returned by the TA
- **INOUT**: Values passed to the TA and returned
Memory Management

- Memory shared between client and TA is _not secure_ (by definition)
  - Can use memory allocated by any mechanism
    - No need for any copies
  - User of the interface allocates all memory directly
    - It is clear who owns all areas of memory
  - User is responsible for freeing all memory
    - The person who allocates it must free it
  - Only one API routine allocates memory
    - TEEC_AllocateSharedMemory
    - Must free this memory using

Memory Ownership

- Operation Starts
- Operation Completes

Client Application
- Owns the buffer
- Trusted Application
- Owns the buffer

Presented by Sequitur Labs Inc. 2016
Memory References
All buffers are represented as memory references

Temporary

```
typedef struct
{
  void* buffer;
  size_t size;
} TEEC_TempMemoryReference;
```

Represented a reference to a user supplied buffer
— Shared only for the length of the call
— On input size is number of valid bytes
— On output size is number of valid bytes
— For output buffers can set size to 0 and get back required size

Registered

```
typedef struct
{
  TEEC_SharedMemory* parent;
  size_t size;
  size_t offset;
} TEEC_RegisteredMemoryReference
```

— Represents a reference to part of a region of shared memory
— Memory is shared for the lifetime of the shared memory structure
— Multiple references can refer to the same shared memory region.

Note: References can overlap
May or may not work depending on

Presented by Sequitur Labs Inc. 2016
Multi-threading

Most operations are thread safe

The exceptions are creation and deletion of objects
  - Only a problem if the same object is used

The effect of writing to the same buffer from multiple TAs is confusing

Can invoke two commands simultaneously on the same TA in the same session
  - The TEE implementation must ensure this does not cause problems

Can access multiple TAs in the same TEE simultaneously
Cancellation

- TA may make commands cancellable
  - Does not have to
  - May choose to ignore cancellation

- Cancellation requires a multi-threaded environment
  - Thread blocking in TA call
  - Second thread to cancel operation

- Set `started` to “0” in operation structure

- Cancel operation by calling `TEEC_CancelOperation`
  - Must be same Operation structure

- Original call may:
  - Complete normally
  - Complete with some error
  - Complete with the `TEEC_ERROR_CANCEL` error code
The TA is assumed to prepend "Hello " to the supplied argument string.

Error handling has been omitted for simplicity.

It should just shut everything down again and exit.

Comments have been omitted.

```c
TEEC_Context context;
TEEC_Session session;
TEEC_Operation operation;

TEEC_Result result;
char *output[50];

result = TEEC_InitializeContext(NULL, &context);
if (result != TEEC_SUCCESS) goto cleanup;

result = TEEC_OpenSession(&context,
&session,
&uuid,
TEEC_LOGIN_PUBLIC,
NULL,
NULL,
NULL);
if (result != TEEC_SUCCESS) goto cleanup1;
```
operation.paramTypes = TEEC_PARAM_TYPES(
    TEEC_MEMREF_TEMP_INPUT,
    TEEC_MEMREF_TEMP_OUTPUT,
    TEEC_NONE,
    TEEC_NONE);
operation.params[0].tmpref.buffer = "World";
operation.params[0].tmpref.size = 5;
operation.params[1].tmpref.buffer = output;
operation.params[1].tmpref.size = sizeof(output);

result = TEEC_InvokeCommand(&session,
    12345,
    &operation,
    NULL);

If (result != TEEC_SUCCESS) goto cleanup2;

printf("%s\n",
    operation.params[1].tmpref.size,
    output);