Using OP-TEE as a Cryptography Engine

Greg Malysa
Timesys Corporation
Outline

- OP-TEE Background
- Adding Crypto Providers in OP-TEE
  - HW RNG
  - HW Crypto Accelerators
- Using OP-TEE Crypto From Linux
  - Using a Custom Library
  - System Integration
  - Standard Solutions
Outline

OP-TEE Background

Adding Crypto Providers in OP-TEE
  HW RNG
  HW Crypto Accelerators

Using OP-TEE Crypto From Linux
  Using a Custom Library
  System Integration
  Standard Solutions
The Internet of Things is Here

Everything is connected now

- Smart Fridges
- Smart TVs
- Industrial/SCADA systems
- Connected medical systems
- In-car networking
- ...

Need to store secret or immutable data: certificates, keys, hashes
How Do We Store Things Securely?

- Hardware Security Module (HSM)
- Trusted Platform Module (TPM)
- Other hardware-based Secure Elements
- On-die Secure Enclaves
- ARM TrustZone
How Do We Store Things Securely?

- Hardware Security Module (HSM)
- Trusted Platform Module (TPM)
- Other hardware-based Secure Elements
- On-die Secure Enclaves
- ARM TrustZone
SoC Security Features

On-die Security Features:
- Tamper-resistant/Secure OTP (SOTP) memory
- Transparent DRAM encryption
- Cryptographic accelerators (e.g. NXP CAAM)

ARM TrustZone:
- Secure Boot
- Secure/non-secure processor states
- Hardware isolation by state
- Secure Monitor Call to request secure action
What is OP-TEE?

The **Open Portable Trusted Execution Environment**.

OP-TEE is built on ARM TrustZone
Why TEE?

- GlobalPlatform specification
- Trusted Applications can run on any TEE
- Linux supports OP-TEE and AMD-TEE
- Secure API Features
  - Encrypted persistent storage
  - Cryptography routines
  - Private data not accessible outside Trusted Application
- Using a TEE offers protection against 0days in kernel, userspace
Outline

- OP-TEE Background
- Adding Crypto Providers in OP-TEE
  - HW RNG
  - HW Crypto Accelerators
- Using OP-TEE Crypto From Linux
  - Using a Custom Library
  - System Integration
  - Standard Solutions
OP-TEE New Platform Bring-up

- Basic platform bringup, booting, running
- Hardware RNG
- Hardware cryptography accelerators
- SOTP and secure peripherals
OP-TEE New Platform Bring-up

- Basic platform bringup, booting, running
- Hardware RNG
- Hardware cryptography accelerators
- SOTP and secure peripherals
Assume some familiarity with

- **core/arch/arm/plat-X/conf.mk**
  - Build configuration and options
  - Controls which features are included in the build
  - Uses make syntax

- **core/arch/arm/plat-X/platform_config.h**
  - Platform specific constants, typically memory layout configuration
  - Regular C header file

- **sub.mk**
  - There’s one in every folder
  - Controls which files are built by adding to `srcs-y`
  - Uses make syntax
RNG Driver Plan

- Understand the crypto RNG API
- Implement driver to match
- Configure build to use driver
The OP-TEE crypto-rng API is only three functions:

\[
\text{TEE\_Result crypto\_rng\_init(const void *data, size\_t dlen)};
\]

\[
\text{TEE\_Result crypto\_rng\_add\_event(enum crypto\_rng\_src sid, unsigned int *pnum, const void *data, size\_t dlen)};
\]

\[
\text{TEE\_Result crypto\_rng\_read(void *buf, size\_t blen)};
\]

- `rng_hw.c` provides `__weak-linked` defaults!
- `crypto_rng_read()` default calls `hw_get_random_byte()`
Basic Driver Implementation

```c
#define HW_RNG_CONTROL 0x00000000
#define HW_RNG_DATA 0x00000004
static vaddr_t hwrng_addr = 0;

TEE_Result crypto_rng_init(const void *data, size_t dlen) {
    hwrng_addr = core_mmu_get_va(HW_RNG_BASE, MEM_AREA_IO_SEC);

    if (!hwrng_addr)
        return TEE_ERROR_GENERIC;

    return TEE_SUCCESS;
}

uint8_t hw_get_random_byte() {
    return io_read8(hwrng_addr + HW_RNG_DATA);
}
```
In platform_config.h, define hardware addresses

```c
#define HW_RNG_BASE 0x00001000
#define HW_RNG_SIZE 0x0000000C
```

In conf.mk, disable the Fortuna PRNG (automatically enabling HW support), and enable new hardware driver

```makefile
$(call force, CFG_WITH_SOFTWARE_PRNG, n)
CFG_MY_HW_RNG := y
```

In sub.mk, include new driver based on config

```makefile
srcs-$(CFG_MY_HW_RNG) += my_hwrng.c
```

In main.c, register HW RNG’s memory and restrict it to the secure state

```c
register_phys_mem_pgdir(MEM_AREA_IO_SEC, HW_RNG_BASE, HW_RNG_SIZE);
```
HW Crypto Accelerators

Start with the internal crypto API, core/crypto/crypto.c

- skcipher, hash, mac all work one way
- akcipher works differently

Hash/mac/skcipher lifecycle:

1. alloc - create hash ctx
2. init - configure ctx
3. update - add data
4. final - finish result
5. free - release memory
crypto_hash_* API uses an ops struct to perform operations

```c
struct crypto_hash_ops {
    TEE_Result (*init)(...);
    TEE_Result (*update)(...);
    TEE_Result (*final)(...);
    void (*free_ctx)(...);
    void (*copy_state)(...);
};
```

Define new ops struct for HW accelerator and connect it!
Life-cycle Operations in \texttt{crypto\_hash\_ops}

Compare hash life-cycle to \texttt{crypto\_hash\_ops}

- \textbf{alloc}: create hash ctx
- \textbf{init}: configure ctx
- \textbf{update}: add data
- \textbf{final}: finish result
- \textbf{free}: release memory

No alloc function in \texttt{crypto\_hash\_ops}
Peek into alloc

```c
TEE_Result crypto_hash_alloc_ctx(void **ctx, uint32_t algo) {
    TEE_Result res = TEE_ERROR_NOT_IMPLEMENTED;
    struct crypto_hash_ctx *c = NULL;
    
    /*
     * Use default cryptographic implementation if no matching
     * drvcrypt device.
     */
    res = drvcrypt_hash_alloc_ctx(&c, algo);
    if (res == TEE_ERROR_NOT_IMPLEMENTED) {
        switch (algo) {
        case TEE_ALG_MD5:
            res = crypto_md5_alloc_ctx(&c);
            break;
            // .. more hash alg cases
        }
    } // ... error checking and result
```
drvcrypt is the interface for integrating hardware accelerators

- Supports one drvcrypt provider for each operation type
- Adding drvcrypt providers requires *no changes* to crypto core
- Some software fallback is automatic
- akcipher (RSA, DSA, ECC, etc.) paths are unique, implement carefully
We only need to implement the HW driver below:

- **HW Driver**
  - HW Init
  - alloc()
  - init()
  - update()
  - final()
  - free_ctx()

- **drvcrypt**
  - drvcrypt_register_*()
  - cipher_alloc()
  - cipher_init()
  - cipher_update()
  - cipher_final()
  - cipher_free_ctx()

- **Crypto API**
  - crypto_*_alloc_ctx()
  - crypto_*_init()
  - crypto_*_update()
  - crypto_*_final()
  - crypto_*_free_ctx()

*Only for skcipher*
- Do regular HW initialization
- `drvcrypt_register_*` used to register a driver for use
  - hash and mac need 1 function to allocate `crypto_*_ctx`
  - skcipher needs instance of `drvcrypt_cipher`
  - akcipher algorithms have unique structs to register
- Regular crypto API will call hardware-associated routines automatically
TEE_Result hw_hash_probe() {
    /*
    * get virtual address of hardware,
    * perform actual hardware initialization,
    * etc. first, then register drvcrypt last
    */
    return drvcrypt_register_hash(hw_hash_alloc);
}

early_init(hw_hash_probe);
HW Alloc Implementation

```c
TEE_Result hw_hash_alloc(struct crypto_hash_ctx **ctx,
    uint32_t algo) {
    struct *hw_hash_ctx = NULL;

    if (!hw_supports_algo(algo))
        return TEE_ERROR_NOT_IMPLEMENTED;

    // don't forget error checking
    hw_hash_ctx = malloc(sizeof(*hw_hash_ctx));
    hw_hash_ctx->algo = algo;
    hw_hash_ctx->ioaddr = hw_get_free_hw();
    hw_hash_ctx->ctx.ops = &hw_hash_ops;

    // later we use container_of to obtain hw_hash_ctx from
    // crypto_hash_ctx
    // just like in the linux kernel
    *ctx = &hw_hash_ctx->ctx
    return TEE_SUCCESS;
}
```
struct hw_hash_ctx {
    struct crypto_hash_ctx ctx;
    uint32_t algo;
    vaddr_t ioaddr;
};

// @todo you have to implement all of these functions
static const struct crypto_hash_ops hw_hash_ops = {
    .init = hw_hash_init,
    .update = hw_hash_update,
    .final = hw_hash_final,
    .free_ctx = hw_hash_free,
    .copy_state = hw_hash_copy_state,
};
In `conf.mk`, enable `drvcrypt`

```
$(call force,CFG_CRYPTO_DRIVER,y)
```

Then, enable sub-drivers for your needs

- `<CFG_CRYPTO_DRV_HASH>`
- `<CFG_CRYPTO_DRV_MAC>`
- `<CFG_CRYPTO_DRV_CIPHER>`
- `<CFG_CRYPTO_DRV_ACIPHER>`

`CFG_CRYPTO_DRV_ACIPHER` has *even more* sub-drivers available

Add driver to `sub.mk`, configure addresses in `platform_config.h`
Outline

- OP-TEE Background
- Adding Crypto Providers in OP-TEE
  - HW RNG
  - HW Crypto Accelerators
- Using OP-TEE Crypto From Linux
  - Using a Custom Library
  - System Integration
  - Standard Solutions
Accessing OP-TEE From Linux

Access options:
- Custom Library
- Linux Kernel Crypto API
- OpenSSL
- PKCS#11

Warning
The code examples to follow are stripped down to highlight specific aspects of the API. They have been stripped to fit on slides and require additional error checking.
Application Flow

- Userspace Application
- TA Interface Library
- TEE Client

Non-secure userspace

- TA_InvokeCommandEntryPoint
- TEE_CipherUpdate

Trusted Application

- crypto_cipher_update
- drvcrypt_cipher_update
- HW Accelerator

OP-TEE Kernel
Trusted Applications (TAs) run in OP-TEE’s userspace

- Use GlobalPlatform TEE Internal Core API
  - Cryptography accessed through “Operation” and “Object” opaque handles

- Interface to Linux applications is RPC through shared memory
What is a TEE Operation?

TEE Operations are opaque references to cryptographic operations

- Operation life-cycle is analogous to hash life-cycle earlier
- Operations are associated with keys using TEE Objects
- Operations can represent any type of cryptographic operation
What are TEE Objects?

TEE Objects are opaque references to data managed by OP-TEE

- 2 flavors of object: Persistent and Transient
- Transient Objects exist only in RAM and for the lifetime of a TA
  - If the TA panics, these are deleted
  - Must initialize before use
  - Recommend using Transient Objects for cryptographic keys
- Persistent Objects store information encrypted at rest
  - Enable long-term storage of sensitive data
  - Stored encrypted on Linux filesystem or in eMMC RPMB
  - Can be restricted to a single TA or shared between TAs
• Each command is identified by a number
• Each command has up to 4 arguments (in OP-TEE)
• Arguments can be inputs/outputs/both
• Arguments can be a 64-bit scalar or a buffer with fixed length
Minimal TA Interface

```c
TEE_Result TA_InvokeCommandEntryPoint(void *session_ctx,
    uint32_t cmd_id, uint32_t param_types,
    TEE_Param params[4]) {

    switch (cmd_id) {
        case CMD_TA_AES_ENCRYPT:
            return TA_EncryptAES(param_types, params);
        case CMD_TA_AES_DECRYPT:
            return TA_DecryptAES(param_types, params);
        default:
            return TEE_ERROR_NOT_SUPPORTED;
    }
}
```
static TEE_ObjectHandle aeskey;

TEE_Result TA_EncryptAES(uint32_t param_types, TEE_Param params[4])
{
    const uint32_t exp_param_types = TEE_PARAM_TYPES(
    TEE_PARAM_TYPE_MEMREF_INPUT, // IV
    TEE_PARAM_TYPE_MEMREF_INPUT, // input
    TEE_PARAM_TYPE_MEMREF_OUTPUT, // output
    TEE_PARAM_TYPE_TYPE_NONE
    );

    if (params != exp_param_types)
        return TEE_ERROR_BAD_PARAMETERS;

    return aes_encrypt(aeskey,
                       params[0].memref.buffer, params[0].memref.size,
                       params[1].memref.buffer, params[1].memref.size,
                       params[2].memref.buffer, &params[2].memref.size);
}
Minimal AES code

```c
TEE_Result aes_encrypt(TEE_ObjectHandle key, const uint8_t *iv,
    size_t ivlen, const uint8_t *in, size_t inlen,
    uint8_t *out, size_t *outlen) {

    TEE_OperationHandle op;
    TEE_Result res;

    res = TEE_AllocateOperation(&op, TEE_ALG_AES_CBC_NOPAD,
        TEE_MODE_ENCRYPT, 128);
    res = TEE_SetOperationKey(op, key);
    TEE_CipherInit(op, iv, ivlen);
    res = TEE_CipherUpdate(op, in, inlen, out, outlen);
    TEE_FreeOperation(op);
    return res;
}
```
Minimal Example Caveats

- Need to check res along the way
- TEE API will **panic** as a safety precaution
- Panic is unrecoverable, do error checking instead!
  - IV length, input and output lengths, block size
- Some other things have been omitted for brevity or to focus on the TEE API
Building a Secure Storage System

- Goal: software-based secure element
- Organize data into slots
- Slots are optionally backed by persistent objects
- Slots support different ops based on type
Storage Application Flow

Library
- open_slot
- encrypt
- decrypt
- genkey
- inject

TA_Invoke

Trusted Application
- CMD_OPEN
- CMD_ENCRYPT
- CMD_DECRYPT
- CMD_GENKEY
- CMD_INJECT
- slot->open
- slot->encrypt
- slot->decrypt
- slot->genkey
- slot->inject
- transient_open
- data_open
- dukpt_open
struct slot_ops {
    // bookkeeping, internal
    TEE_Result (*new)(struct slot *);
    TEE_Result (*init)(struct slot *);
    TEE_Result (*write)(struct slot *);

    // actual operations interface
    TEE_Result (*encrypt)(struct slot *, ...);
    TEE_Result (*decrypt)(struct slot *, ...);
    // ...
};
Opening a Slot

TEE_Result __open_slot(uint32_t slot) {
    uint32_t flags = TEE_DATA_FLAG_ACCESS_READ |
                    TEE_DATA_FLAG_ACCESS_WRITE;
    struct slot *slot = &slots[slot];
    TEE_Result res;

    res = slot->ops->new(slot);
    res = TEE_OpenPersistentObject(TEE_STORAGE_PRIVATE, slot->id,
                                   strlen(slot->id), flags, &slot->persistent);
    if (!res)
        res = slot->ops->init(slot);
    if (res == TEE_ERROR_ITEM_NOT_FOUND) {
        res = TEE_SUCCESS;
        slot->persistent = TEE_HANDLE_NULL;
    }
    return res;
}
static uint32_t write_flags = TEE_DATA_FLAG_ACCESS_READ |
    TEE_DATA_FLAG_ACCESS_WRITE | TEE_DATA_FLAG_ACCESS_WRITE_META |
    TEE_DATA_FLAG_OVERWRITE;

TEE_Result transient_write(struct slot *slot) {
    return TEE_CreatePersistentObject(
        TEE_STORAGE_PRIVATE, slot->id, strlen(slot->id), write_flags,
        slot->transient, NULL, 0,
        &slot->persistent);
}

TEE_Result data_write(struct slot *slot) {
    return TEE_CreatePersistentObject(
        TEE_STORAGE_PRIVATE, slot->id, strlen(slot->id), write_flags,
        TEE_HANDLE_NULL, slot->data, slot->length,
        &slot->persistent);
}
Remaining TA Pieces

- Routine tasks
  - Fill in ops struct per slot type
  - Fill in switch statement for invoke
  - Do error checking and parameter validation

- One time programming support (for mfg)
- Handle TEE_ERROR_CORRUPT_OBJECT
Library mirrors the invoke structure
- At least one library function per command
- Additional library functions for parameterized commands, such as “AES OP” with ENCRYPT or DECRYPT as an argument

Typical TEE Client-based implementation
- Disable HW crypto drivers that are now owned by OP-TEE
- Register new crypto providers with the kernel (sounds familiar)
  - `crypto_register_alg(struct crypto_alg *)`
  - `crypto_register_shash(struct shash_alg *)`
- Overall implementation is analogous to userspace library
struct cipher_alg {
    unsigned int cia_min_keysize;
    unsigned int cia_max_keysize;
    int (*cia_setkey)(struct crypto_tfm *tfm, const u8 *key,
        unsigned int keylen);
    void (*cia_encrypt)(struct crypto_tfm *tfm, u8 *dst,
        const u8 *src);
    void (*cia_decrypt)(struct crypto_tfm *tfm, u8 *dst,
        const u8 *src);
};
OpenSSL Integration

- OpenSSL can be extended through custom engines
- An engine could wrap our library
- Easy path to integration?
OpenSSL Integration

- OpenSSL can be extended through custom engines
- An engine could wrap our library
- Easy path to integration?
  - No
- Consider alternatives:
  - AF_ALG and kernel interface
  - Use OpenSC’s PKCS#11 wrapper engine, libp11
  - Write PKCS#11 compliant provider
The PKCS#11 specification has everything we want in key storage:

- Multiple slots for cryptographic objects
- Interface for performing operations on those objects
- Cryptographic data is not exposed to the user
- Cryptoki API also bears some resemblance to TEE APIs

Lots of other applications already work with PKCS#11 providers
OP-TEE team has been working on pkcs11 TA and libckteec
Not quite ready but very promising
See Ruchika and Etienne’s presentation from LVC21 earlier this year
fTPM

- Software implementation of TPM
- Microsoft provides an OP-TEE-backed implementation
- Mainline Linux driver available for access
- Can use directly or plug into PKCS#11 adapter
Use OP-TEE to provide cryptographic services
Add hardware accelerators directly to OP-TEE
Access those services through Trusted Applications
Access Trusted Applications through a userspace library
Be on the look out for pkcs11 and libckteec to avoid rerolling them!