The Android fingerprint system

Andrew Lee-Thorp

Synopsys Software Integrity Group

About Me

Andrew Lee-Thorp

@Synopsys Software Integrity Group (from Coverity, Codenomicon and others + Cigital)

Android assessments, tool development, system design and development, code review, cutting code, threat modelling

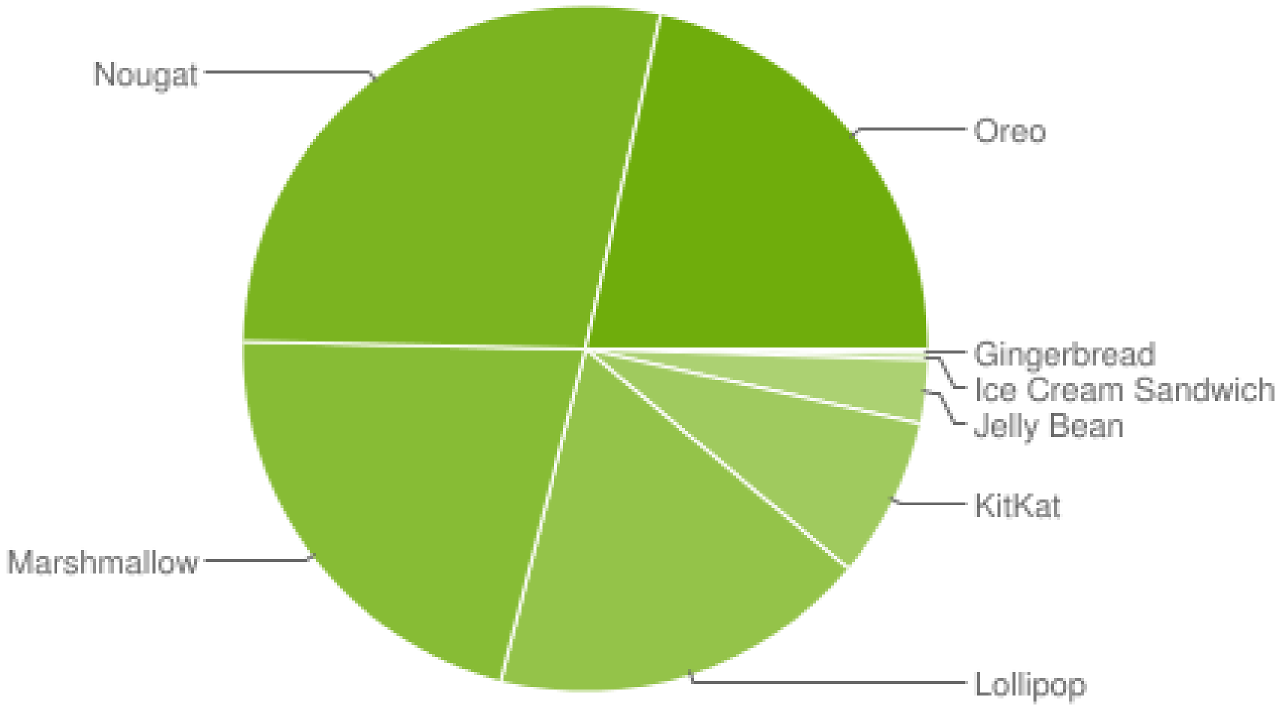
> 10 yrs cutting code, nearly 10 yrs in security

Why this talk?

Agenda

- Introduction
- Framework APIs and the developer perspective
- Fingerprint for local authentication
- Using fingerprint to unlock hardware-backed keys
- Fingerprint spoofing
- A journey from normal world to secure world and a short detour into TrustZone
- What can go wrong?

Fingerprint HAL and requirements introduced in Marshmallow (6.0)



Figures on 26 October, 2018

> 70% on Android 6.0 or later

Benefits

- Streamlined user authentication
  - Performance and usability benefit
  - “Single sign-on”
- Previously: “alignment” with iPhone, iPad

Stakeholders and entities

- End users: am I secure?
- Organisations, e.g. banks: will I put my users at risk?
  - Corollary: will I suffer any reputational damage?
  - How does fingerprint compare to PIN?
  - How ~~does~~ did it compare to touchid (iOS)?
  - Is fingerprint as a second factor?
- Security researchers:
  - How can I attack the system?
- Device manufacturer, e.g. Samsung, Sony
- System-on-chip vendor, e.g. Qualcomm, HiSilicon, MediaTek
- Fingerprint sensor vendor, e.g. FPC, Synaptics, Goodix
- Google & AOSP

Risks

- Physical
  - Fingerprint Forgery (aka spoofing)
- Application risks
  - Misuse of the APIs
- Multiple technical risks due to the implementation
  - Multiple critical hw and sw components are involved, each provided by different 3rd party (vendor)
  - 3rd Party ==> proprietary, non-standard, unknown
  - Any one component failure “could” compromise the overall security
  - Built on TrustZone
- Disclosure of fingerprint material (image) or derived material (template)
  - Bugs
  - Backdoors
- Bypass Fingerprint Authentication
  - Trick the app
  - Trick the system
  - Trick the user (“clickjacking” / confused deputy attack)

Android Compatibility Definition Document

Enumerates the requirements that must be met

7.3.10.1. Fingerprint Sensors

[SR] STRONGLY RECOMMENDED that spoof and imposter acceptance rate not > 7%.

[C-1-5] MUST rate limit attempts for at least 30 seconds after five false trials

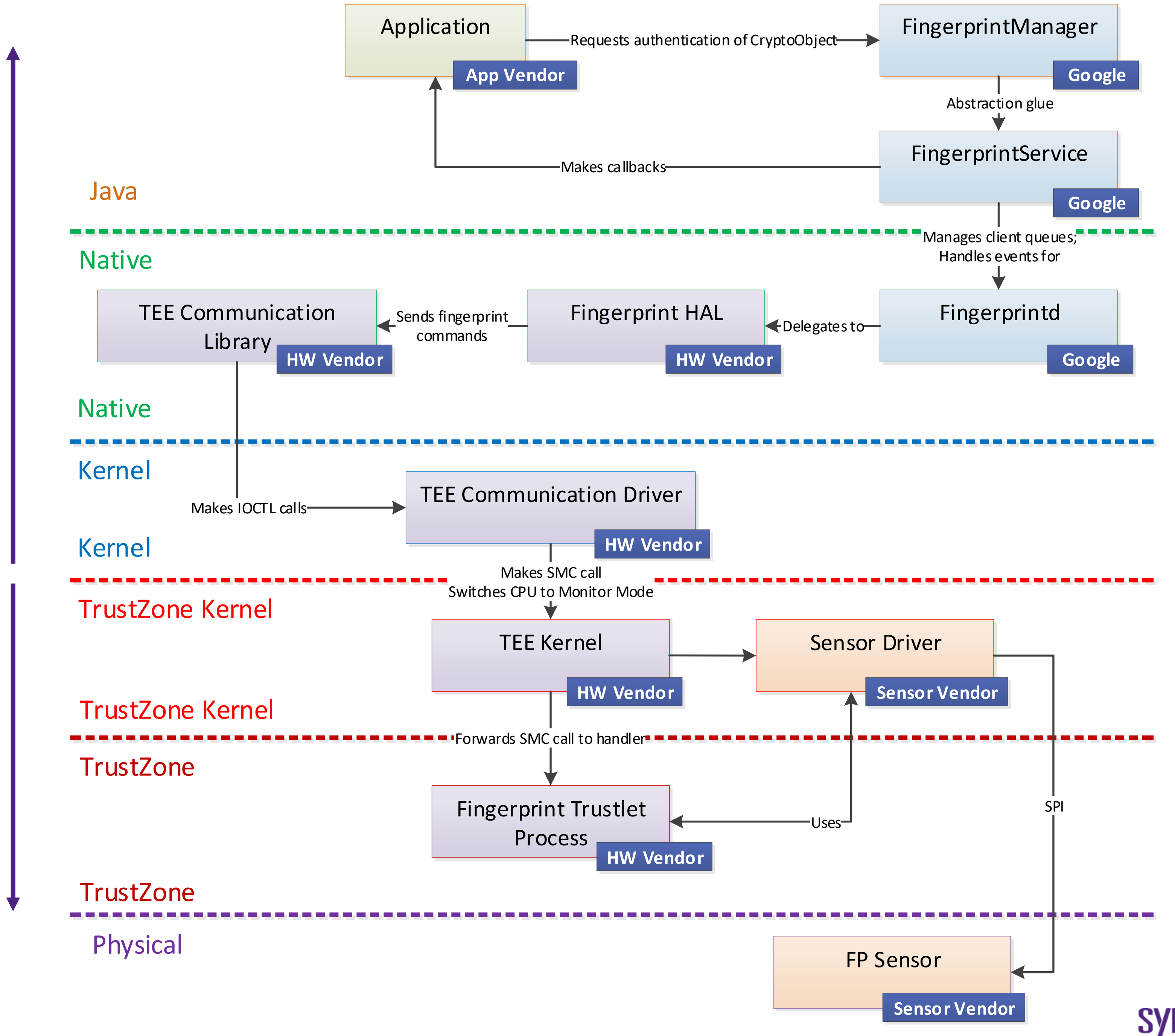
[C-1-8] MUST first establish a chain of trust by first establishing PIN/pattern/password that's secured by TEE.

[C-1-2] MUST fully implement the corresponding API as described in the Android SDK documentation.

System Architecture

“Normal” world (REE / Nwd)

“Secure” world (Trusted / Swd)



Framework support

Step 1: check for hardware support and device security enabled

```
<uses-permission android:name="android.permission.USE_FINGERPRINT" />


if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
        enabledFingerprints = true;
}

FingerprintManager.isHardwareDetected()

FingerprintManager.hasEnrolledFingerprints()

KeyguardManager.isDeviceSecure()
```

Step 2: request authentication, provide callback

```
FingerprintManager.authenticate(cryptObject, cancellationSignal, flags,
callback, handler)
```

Step 3: Handle result

```
public class MyCallback extends FingerprintManager.AuthenticationCallback {

    @Override
     public void onAuthenticationError(…) {
    }

    @Override public void onAuthenticationHelp(…){
    }

    @Override public void onAuthenticationFailed() {
    }

    @Override public void
    onAuthenticationSucceeded(FingerprintManager.AuthenticationResult result) {
        // result contains the cryptObject
```

BiometricPrompt (Android Pie)

Standardises the UI, changes the flow

```
@Override
 public void onAuthenticationError(…) {
}
```

`BIOMETRIC_ERROR_NO_BIOMETRICS` - no fingerprint (biometric) enrolled.
`BIOMETRIC_ERROR_HW_NOT_PRESENT` - fingerprint (biometric) sensor not present
`BIOMETRIC_ERROR_HW_UNAVAILABLE`

```
<uses-feature
   android:name="android.hardware.fingerprint" android:required="true" />
```

Google play store feature (unrelated to Biometric Prompt)

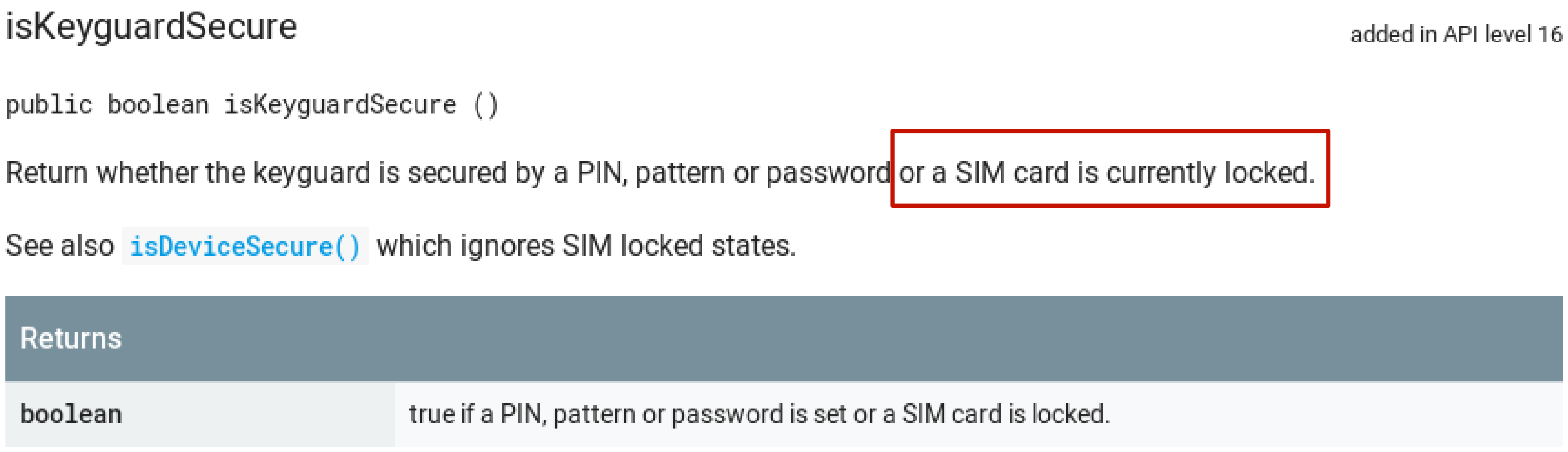
API details are hidden in the fine print

https://github.com/googlesamples/android-FingerprintDialog

Do not use isKeyguardSecure.

```
 if (!keyguardManager.isKeyguardSecure()) {
            // Show a message that the user hasn't set up a fingerprint or
lock screen.
            Toast.makeText(this,
                    "Secure lock screen hasn't set up.\n"
                            + "Go to 'Settings -> Security -> Fingerprint' to
set up a fingerprint",
                    Toast.LENGTH_LONG).show();
            purchaseButton.setEnabled(false);
            purchaseButtonNotInvalidated.setEnabled(false);
            return;
        }
```

API details are in the fine print



isKeyguardSecure

added in API level 16

```
public boolean isKeyguardSecure ()
```

Return whether the keyguard is secured by a PIN, pattern or password or a SIM card is currently locked.

See also `isDeviceSecure()` which ignores SIM locked states.

Returns	
`boolean`	true if a PIN, pattern or password is set or a SIM card is locked.

3. Authentication flow options

1

```
@Override public void
onAuthenticationSucceeded(FingerprintManager.AuthenticationResult result) {
    // ignore cryptObject
     authenticationSucceeded = true;
```

2

```
@Override public void
onAuthenticationSucceeded(FingerprintManager.AuthenticationResult result) {
    // use the cryptoObject on locally encrypted data (e.g. shared prefs)
     tryDecrypt(result.getCryptoObject().getCipher());
```

3

```
@Override public void
onAuthenticationSucceeded(FingerprintManager.AuthenticationResult result) {
    // use the cryptoObject to sign server challenge
     // really remote authentication
     trySignChallenge(result.getCryptoObject().getCipher(), challenge);
```

Local authentication

App "authenticates" user using device credentials – fingerprint, PIN, …

Bypassing local authentication

Local authentication is using fingerprint verification to authorize an action locally

```
public class FingerprintUiHelper extends
FingerprintManager.AuthenticationCallback {

   @Override public void
   onAuthenticationSucceeded(FingerprintManager.AuthenticationResult result) {
       // result contains the cryptObject
```

Poor integration of the authentication flow can be bypassed by **privileged** attacker.

Attacking local authentication

https://github.com/googlesamples/android-FingerprintDialog and frida

Fingerprints and the Keystore

TEE Backed Keystore

Conformant fingerprint implementations, for devices released with Android 6.0 or later, must make use of a hardware-backed keystore (i.e. keystore backed by a TEE).

Key material + operations stay inside secure hardware

```
SecretKey sk = keyGenerator.generateKey();
SecretKeyFactory factory =
   SecretKeyFactory.getInstance(sk.getAlgorithm(), "AndroidKeyStore");
KeyInfo keyInfo = (KeyInfo) factory.getKeySpec(sk, KeyInfo.class);
if (keyInfo.isInsideSecureHardware()) {
   enabledFingerprints = true;
}
```

We found that some key combinations are software-backed – inconsistent across devices

Key Use Bound to Authentication Event

- Unlock key in AndroidKeyStore using fingerprint.

```
KeyGenerator keyGenerator = KeyGenerator.getInstance("AES",
"AndroidKeyStore");
keyGenerator.init(new KeyGenParameterSpec.Builder(keyAlias,
   KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
    .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
    .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
    .setUserAuthenticationRequired(true)
    .setUserAuthenticationValidityDurationSeconds(60)
    .setUserAuthenticationValidWhileOnBody(false)
    .setInvalidatedByBiometricEnrollment(true) // SDK >= 24
.build());
SecretKey key = keyGenerator.generateKey();
```

KEY REMAINS UNLOCKED FOR 60 seconds

NOT RECOMMENDED

“By *default*, if user authentication is required, it must take place for every use of the key. “
https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.Builder#setUserAuthenticationValidityDurationSeconds(int)

```
keyGenerator.init(new KeyGenParameterSpec.Builder(keyAlias,
                KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
    .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
    .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
    .setUserAuthenticationRequired(true) *
    .setUserAuthenticationValidityDurationSeconds(60) **
    .setUserAuthenticationValidWhileOnBody(false)
```

In reality …

- “Requiring authentication” has no effect unless **key duration is in [-1,0]**
- Device just needs to be unlocked.

Attacker enrols new fingerprint, tries to authorises use of the key

```
// SDK >= 24
KeyGenParameterSpec.Builder.setInvalidatedByBiometricEnrollment(true)
```

INVALIDATE KEY ON NEW ENROLLMENT

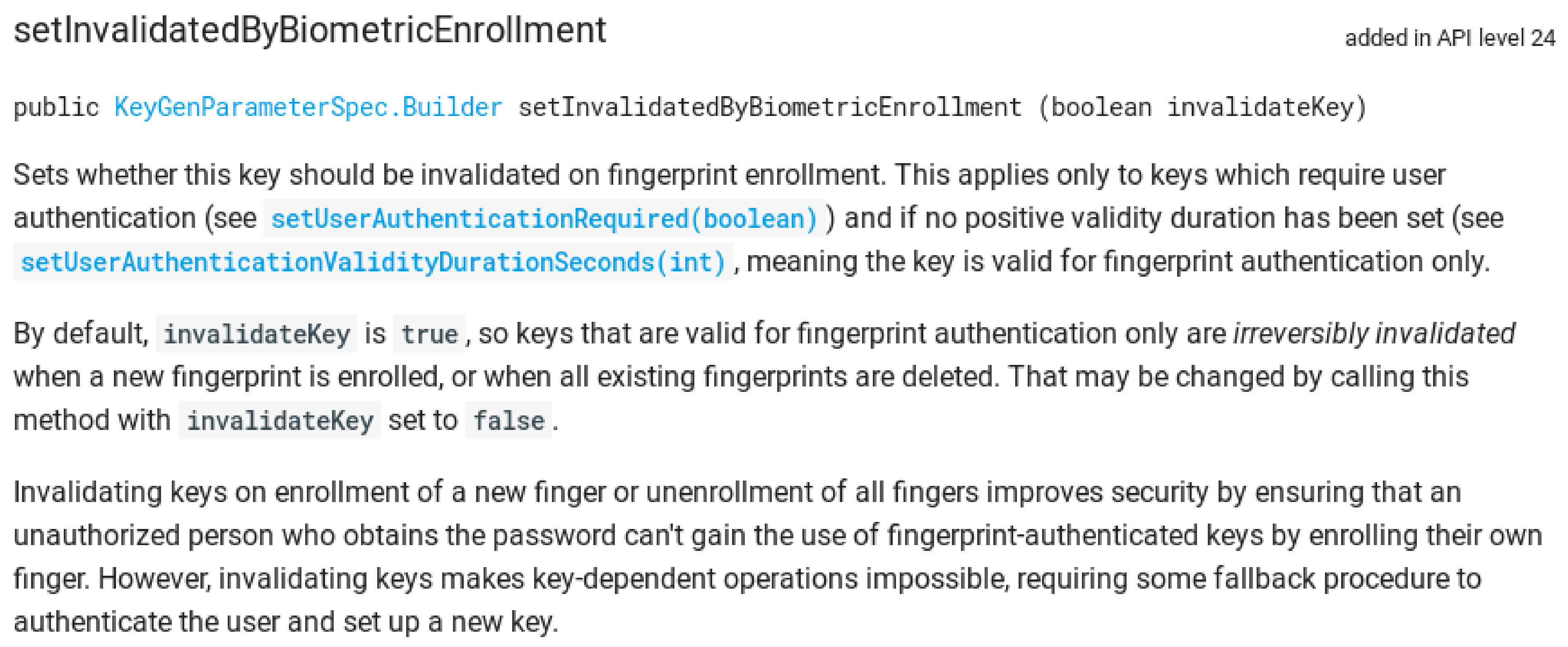
What if the KeyGenerator specification is “bad”?

… then what should happen below?

```
KeyGenerator keyGenerator = KeyGenerator.getInstance("AES",
"AndroidKeyStore");
keyGenerator.init(new KeyGenParameterSpec.Builder(keyAlias,
   KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
   .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
   .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
   .setUserAuthenticationRequired(true)
   .setUserAuthenticationValidityDurationSeconds(60)
   .setUserAuthenticationValidWhileOnBody(false)
   .setInvalidatedByBiometricEnrollment(true)
.build());
SecretKey key = keyGenerator.generateKey(); // ???
```

Documentation is silent

https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.Builder#setInvalidatedByBiometricEnrollment(boolean)



setInvalidatedByBiometricEnrollment

added in API level 24

```
public KeyGenParameterSpec.Builder setInvalidatedByBiometricEnrollment (boolean invalidateKey)
```

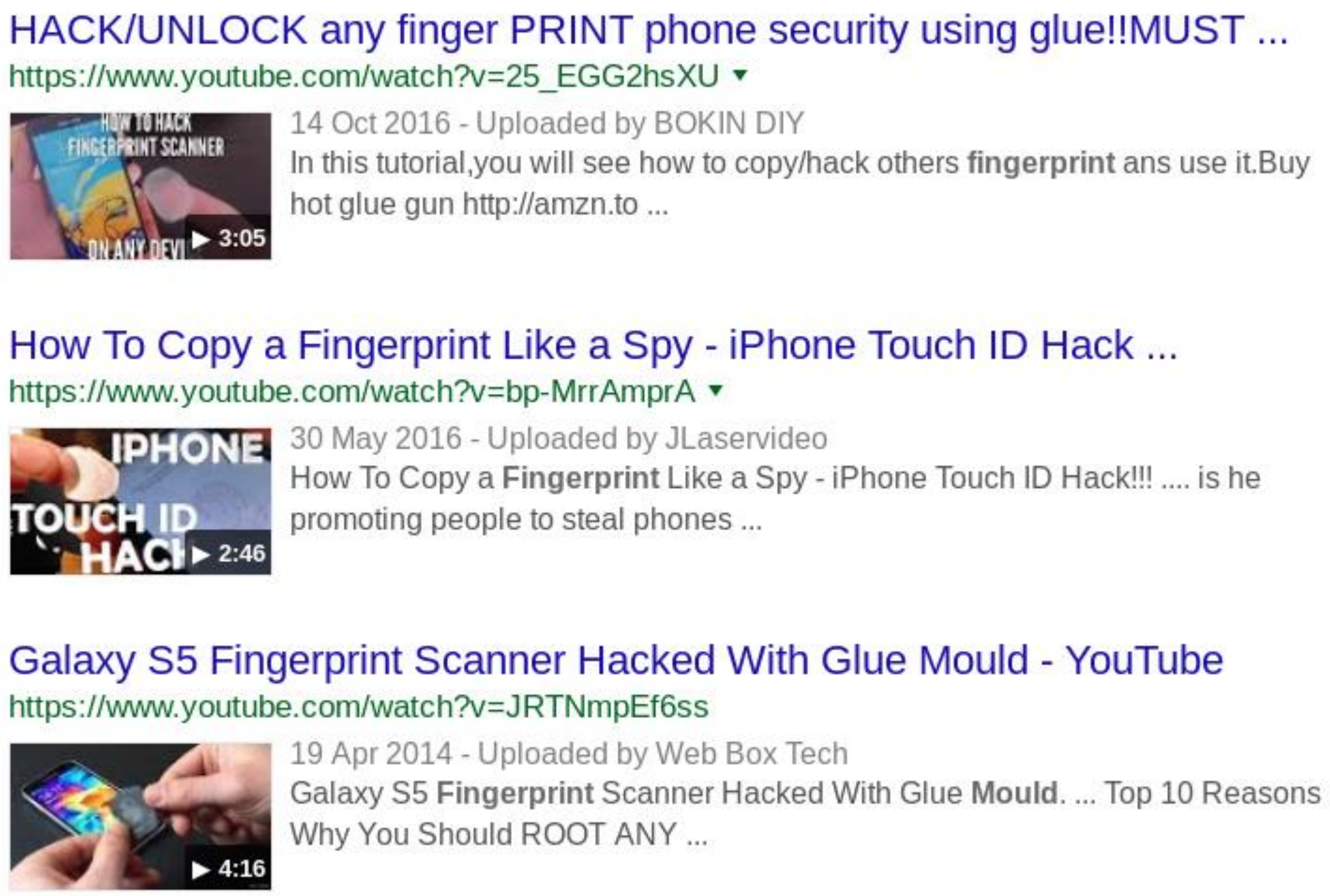
Sets whether this key should be invalidated on fingerprint enrollment. This applies only to keys which require user authentication (see `setUserAuthenticationRequired(boolean)`) and if no positive validity duration has been set (see `setUserAuthenticationValidityDurationSeconds(int)`, meaning the key is valid for fingerprint authentication only.

By default, `invalidateKey` is `true`, so keys that are valid for fingerprint authentication only are *irreversibly invalidated* when a new fingerprint is enrolled, or when all existing fingerprints are deleted. That may be changed by calling this method with `invalidateKey` set to `false`.

Invalidating keys on enrollment of a new finger or unenrollment of all fingers improves security by ensuring that an unauthorized person who obtains the password can't gain the use of fingerprint-authenticated keys by enrolling their own finger. However, invalidating keys makes key-dependent operations impossible, requiring some fallback procedure to authenticate the user and set up a new key.

Fingerprint Spoofing

Fingerprint Spoofing



HACK/UNLOCK any finger PRINT phone security using glue!!MUST ...
https://www.youtube.com/watch?v=25_EGG2hsXU ▾
14 Oct 2016 - Uploaded by BOKIN DIY
In this tutorial,you will see how to copy/hack others **fingerprint** ans use it.Buy hot glue gun http://amzn.to ...

How To Copy a Fingerprint Like a Spy - iPhone Touch ID Hack ...
https://www.youtube.com/watch?v=bp-MrrAmprA ▾
30 May 2016 - Uploaded by JLaservideo
How To Copy a **Fingerprint** Like a Spy - iPhone Touch ID Hack!!! .... is he promoting people to steal phones ...

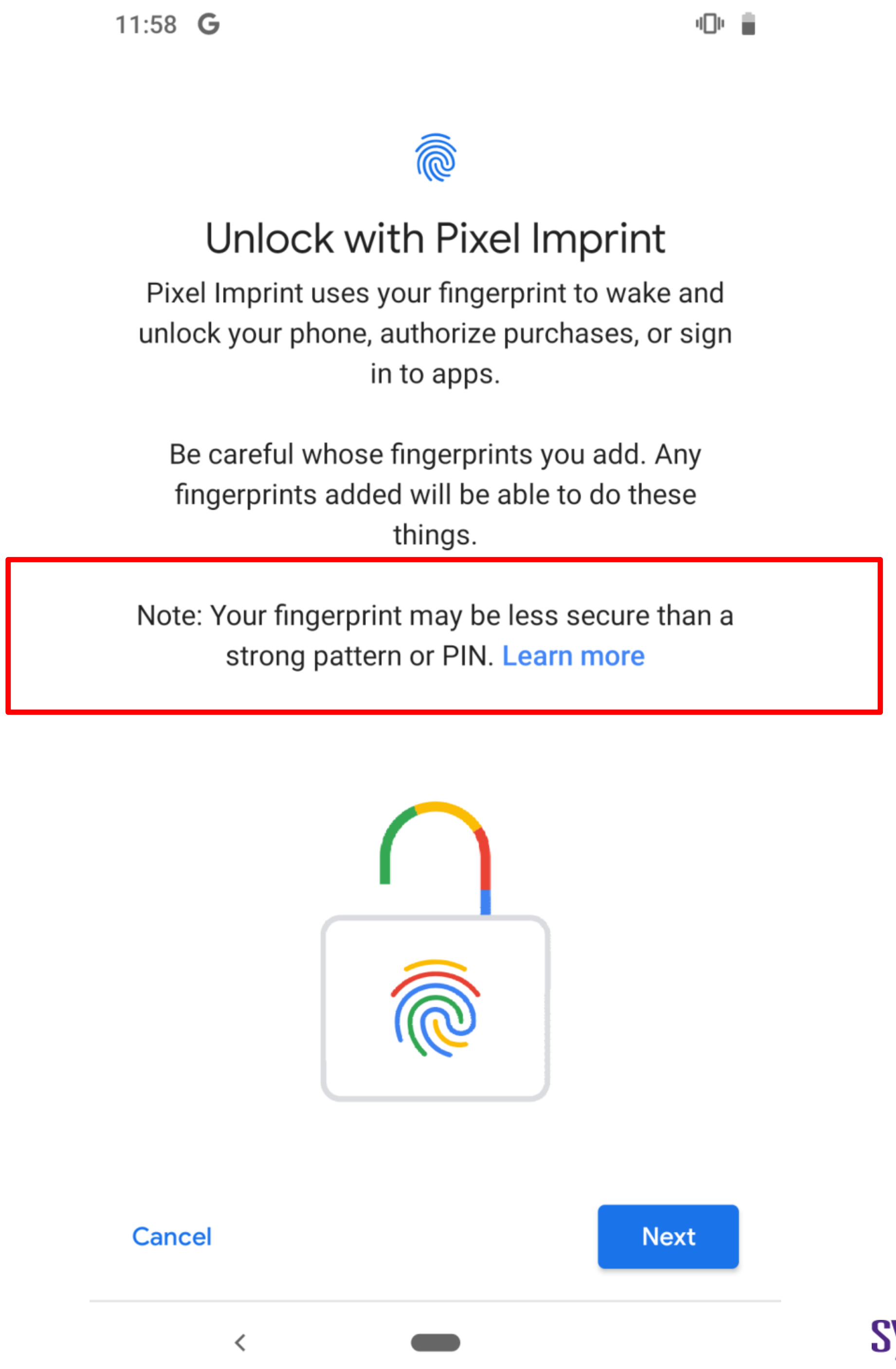
Galaxy S5 Fingerprint Scanner Hacked With Glue Mould - YouTube
https://www.youtube.com/watch?v=JRTNmpEf6ss
19 Apr 2014 - Uploaded by Web Box Tech
Galaxy S5 **Fingerprint** Scanner Hacked With Glue **Mould**. ... Top 10 Reasons Why You Should ROOT ANY ...

Spoofing and the CDD

- [SR] Are STRONGLY **RECOMMENDED** to have a spoof and imposter acceptance rate not higher than **7%**

- [C-1-4] **MUST disclose** that this mode may be less secure than a strong PIN, …, and **clearly enumerate the risks** of enabling it, **if** the spoof and imposter acceptance rates are higher than 7% (Introduced Android 8.1 CDD)

The Spoof Accept Rate (SAR) is the "chance that a biometric model accepts a previously recorded, known good sample". For fingerprint sensors, using a mould of an enrolled fingerprint to unlock a user's phone counts as a spoof attack


11:58 G

Unlock with Pixel Imprint

Pixel Imprint uses your fingerprint to wake and unlock your phone, authorize purchases, or sign in to apps.

Be careful whose fingerprints you add. Any fingerprints added will be able to do these things.

Note: Your fingerprint may be less secure than a strong pattern or PIN. Learn more

SYNOPSYS®

Grandma’s recipe

Step One: For the creation of **moulds** for casting fingerprints a negative (-ve) fingerprint impression is first taken in a suitable moulding material such as Sugru/Pratley Putty as outlined below:

1. Form a ball from mould material.
2. Press firmly onto to mould material.
3. Leave to cure until non-pliable.

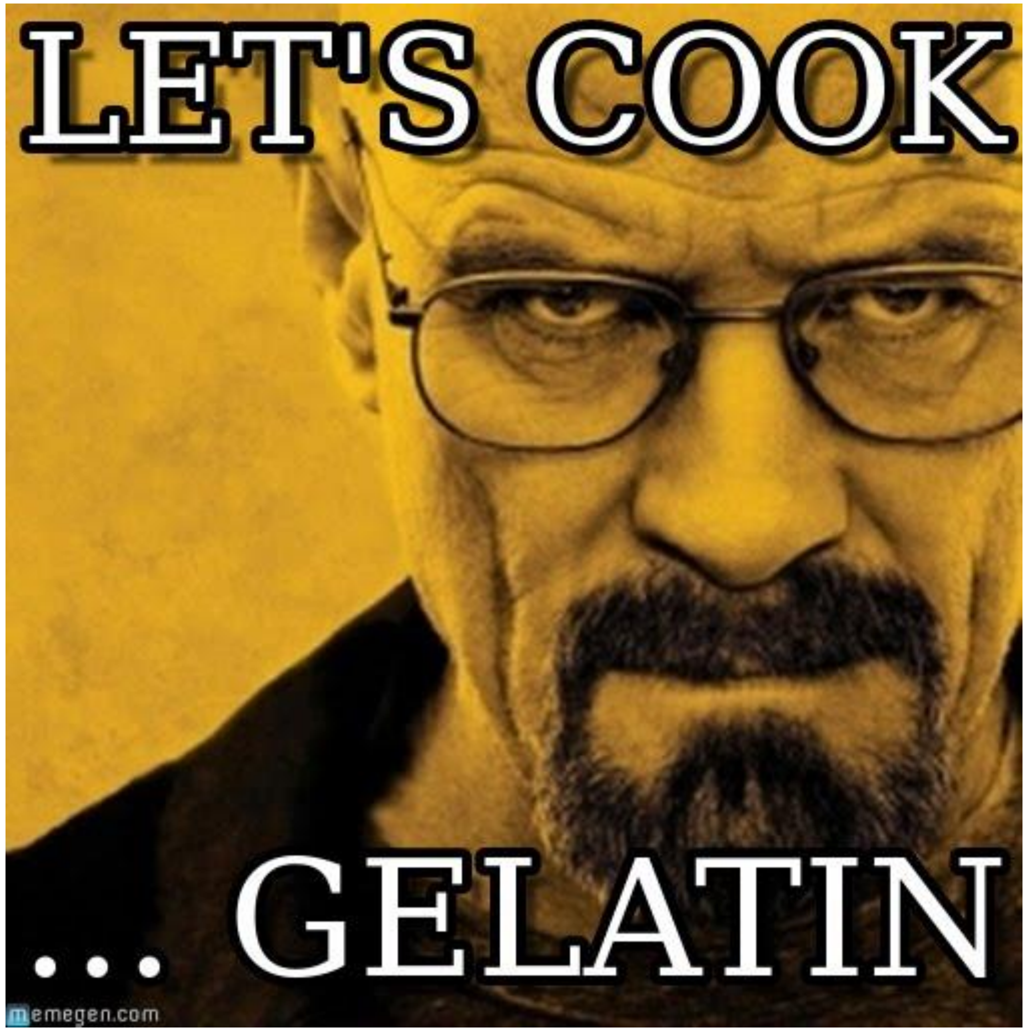
Steps two and three

Step Two: Create the **cast material** by heating gelatin powder and water as shown below:

1. Add gelatin powder to water in a ratio approximately between 0.8:1 and 1:1 volume parts of gelatin to water.
2. Warm at low heat, stirring constantly.
3. Do **not** boil.
4. Remove from the heat and leave to cool.

Step Three:

1. Pre-conditions (a) the mould has set/hardened and (b) the casting material is now viscous but cooled.
2. Pour the casting material into the mould and leave to set.


LET'S COOK
... GELATIN
memegen.com

A short spoofing video

Logs can help identify good models

```
03-10 04:04:32.188  1202  1359 D fpc_tac : capture image start
03-10 04:04:32.330  1202  1359 D fpc_tac : capture image end
03-10 04:04:32.330  1202  1359 D fpc_fingerprint_hal: fpc_notificator: type 2
03-10 04:04:32.330  1202  1359 D fpc_hidl: onAcquired(code=6, vendor=1)
03-10 04:04:32.330  1202  1359 D libfingerprint_core: _push_message: message passed
03-10 04:04:32.332  1921  1921 W FingerprintManager: Invalid acquired message: 6, 1
03-10 04:04:32.344  1202  1359 D fpc_tac : IDD_identify_stat KPI Capture time: 0 ms Identify time: 13
ms
03-10 04:04:32.350  1202  1359 D fpc_tac : IDD_identify_stat: hwid 0x711 res: 1 cov: 93, qual: 33,
covered zones: 4095, score: 93, idx: 0
03-10 04:04:32.352  1202  1359 D fpc_fingerprint_hal: fpc_notificator: type 2
03-10 04:04:32.352  1202  1359 D fpc_hidl: onAcquired(code=0, vendor=0)
03-10 04:04:32.352  1202  1359 D libfingerprint_core: _push_message: message passed
03-10 04:04:32.355  1202  1359 D fpc_fingerprint_hal: fpc_notificator: type 1
03-10 04:04:32.355  1202  1359 D libfingerprint_core: task_set_tryagain: task 'authenticate' restart = 0
03-10 04:04:32.355  1202  1359 D fpc_hidl: onAuthenticated(fid=-1435385119, gid=0)
```

[C-1-9] MUST NOT enable 3rd-party applications to distinguish between individual fingerprints.

[System -> Kernel] -> [TEE]

A journey from normal world to secure world

fingerprintd

https://android.googlesource.com/platform/system/core/+/nougat-release/fingerprintd/fingerprintd.cpp

```
int main() {
    ALOGI("Starting " LOG_TAG);
    android::sp<android::IServiceManager> serviceManager =
    android::defaultServiceManager();
    android::sp<android::FingerprintDaemonProxy> proxy =
        android::FingerprintDaemonProxy::getInstance();
    android::status_t ret = serviceManager->addService(
        android::FingerprintDaemonProxy::descriptor, proxy);
...
/*
* We're the only thread in existence, so we're just going to process
* Binder transaction as a single-threaded program.
*/
    android::IPCThreadState::self()->joinThreadPool();
    ALOGI("Done");
    return 0;
```

1. **Export a binder**
2. **Load HAL implementation**
3. **Proxy binder calls via in-process proxy to HAL implementation**

HAL implementation

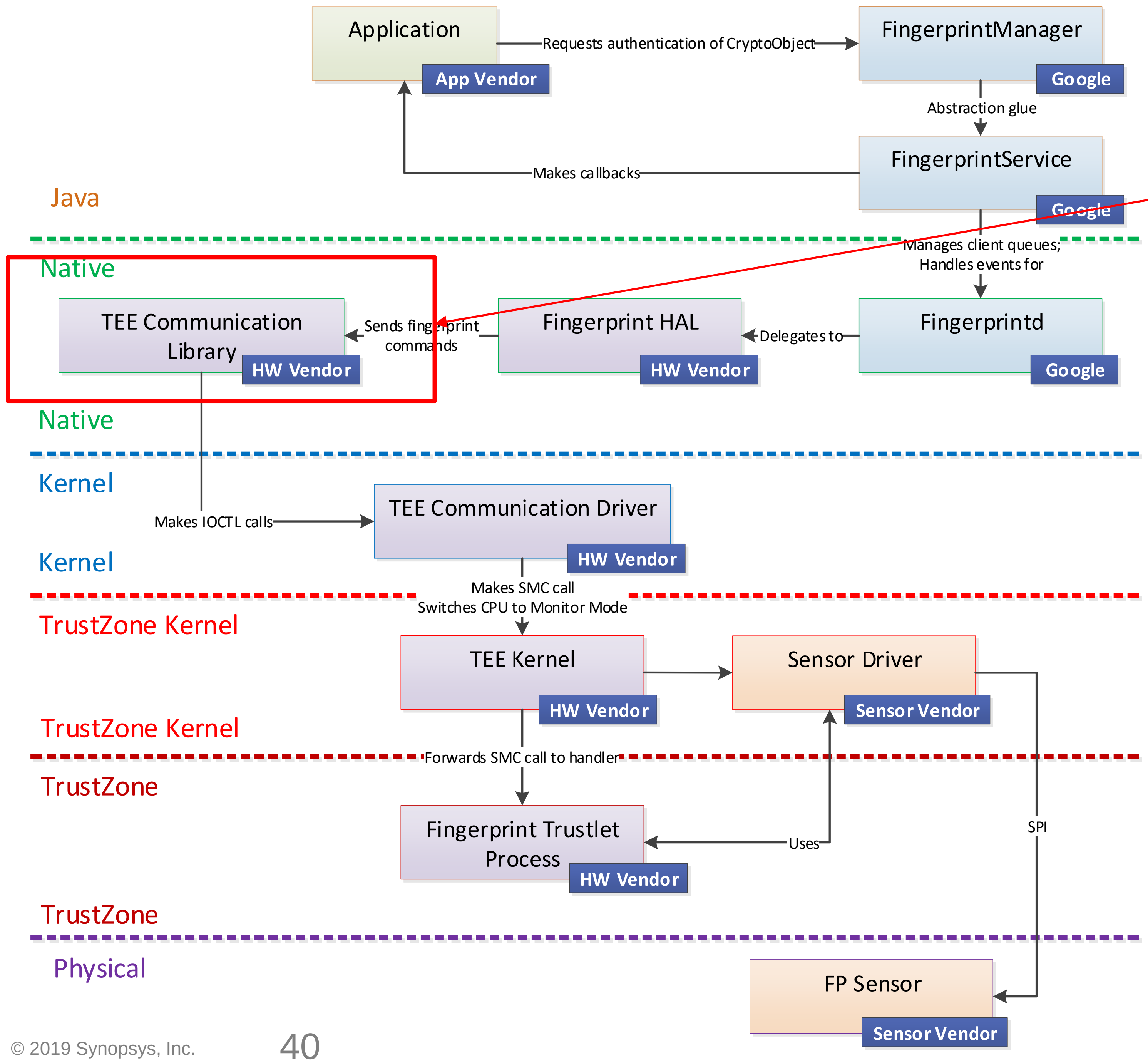
1. Implements the functions in libhardware/include/hardware/fingerprint.h
2. Communicate with the device specific hardware
3. Is closed source

… but we can reverse it and identify HAL functionality, trustlet command ids, and extended functionality

```
__int64 __fastcall BAuth_Enroll_Init(int *a1, __int64 a2, unsigned int a3, int a4,
_DWORD *a5, int a6, __int64 a7)
{
...
  __android_log_print(3LL, "bauth_TLC_Communicator", "Call FP cmd 0x%x", 2LL);
  if ( !gVFMQSEEHandle )
  {
  ...
  v19 = QSEECom_send_cmd(*(_QWORD *)
    gVFMQSEEHandle, v15, 384LL, v16, 64LL);
```

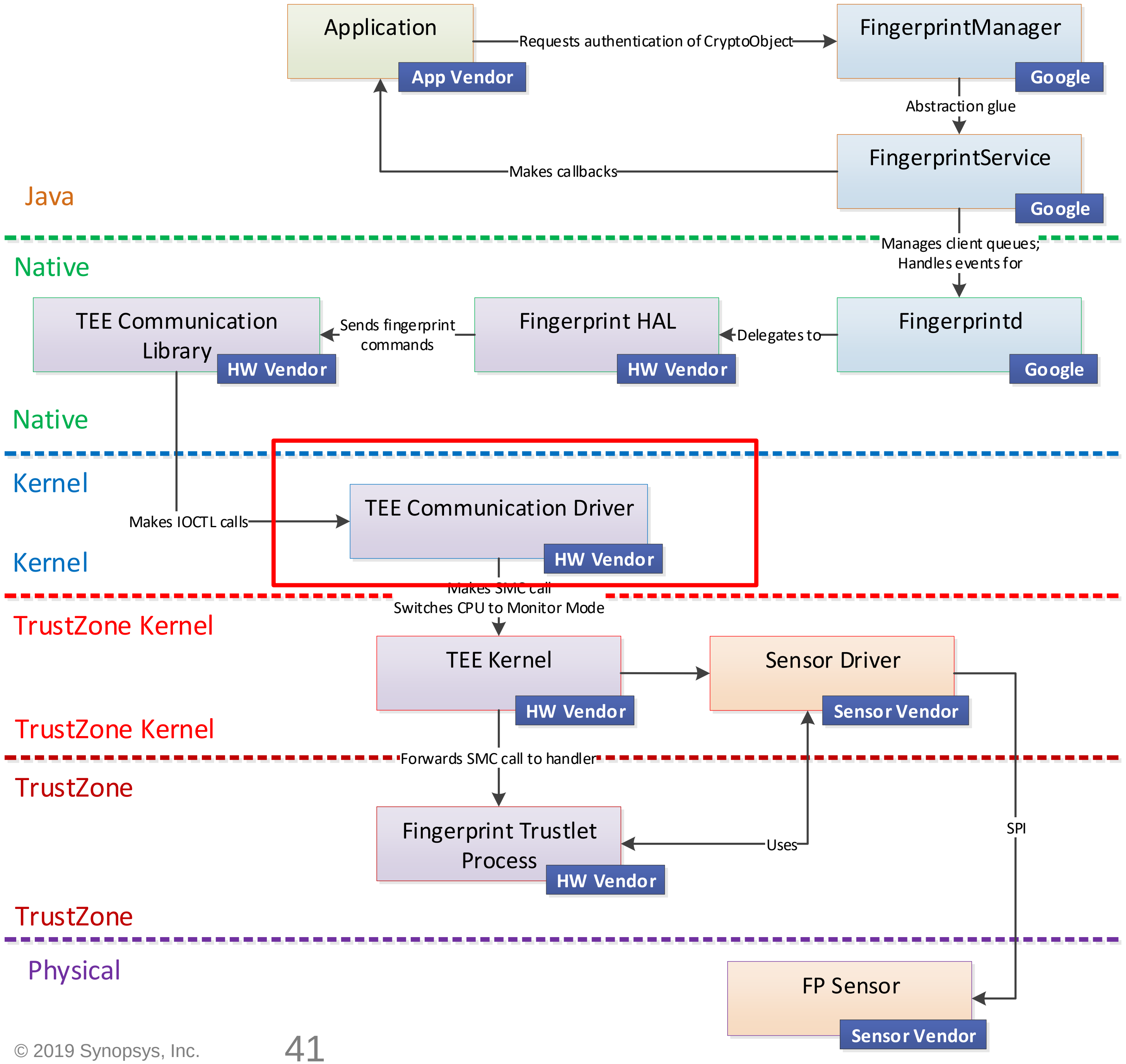
```
D bauth_FPBAuthService: fpop : 100023
D fingerprintd: onAcquired(10004)
D bauth_TLC_Communicator: Call FP cmd 0x2
D bauth_TLC_Communicator: Check the Trustlet return code is
completed
D bauth_FPBAuthService: check_opcode status = 1, opcode = 0,
func_ret_val = 0, function_status = 0, timeout = 0
D bauth_FPBAuthService: FPBAuthService, 1032
```

… and correlate the command ids with the trustlet and logs



Normal-world TEE Communication Library

1. User-space shared object
2. Closed-source, developed by SoC vendor
3. Abstracts IOCTL commands over kernel virtual device

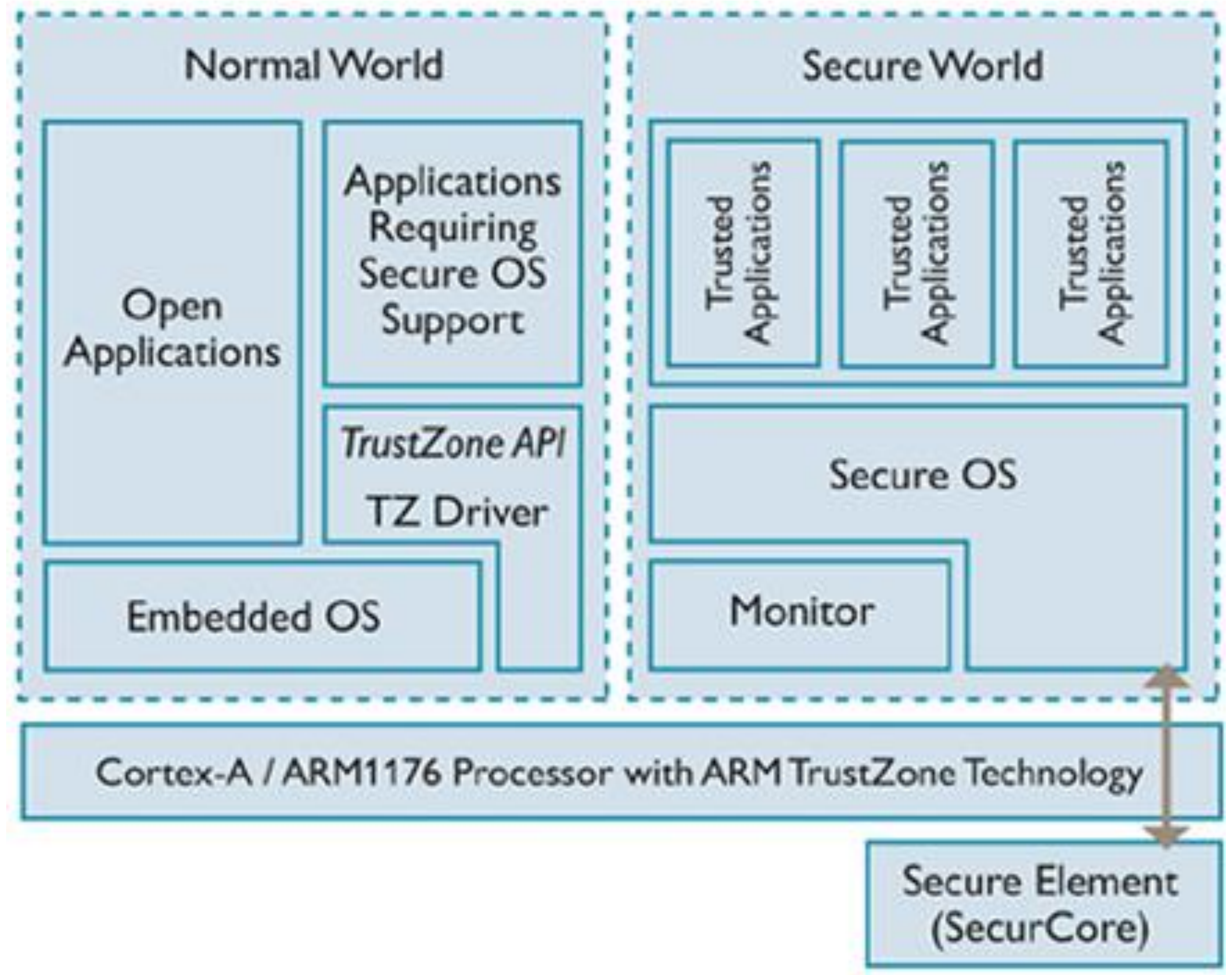


Kernel TEE communication module

1. Protected kernel virtual device
2. Switches to secure world using Secure Monitor Call

/dev/qsee, /dev/mobicore, /dev/tc_ns_client

A very short detour on some aspects of TrustZone and TEEs



Exception Levels

3.2.2 Exception levels

The Armv8 exception model defines exception levels EL0-EL3, where:

- EL3 provides support for a Secure state, see *3.2.3 Security state*.

Typical exception level usage model

- EL0 Applications.
- EL1 OS kernel and associated functions that are typically described as privileged.
- EL2 Hypervisor.
- **EL3 Secure monitor.**

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0488c/CHDDAHFB.html, A73, e.g. A8+

Security state

3.2.3 Security state

Secure state
In Secure state, the processor:

- Can access both the Secure memory address space and the Non-secure memory address space.
- When executing at EL3, can access all the system control resources.

Non-secure state
In Non-secure state, the processor:

- Can access only the Non-secure memory address space.
- Cannot access the Secure system control resources.

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0488c/CHDDAHFB.html, A73, e.g. A8+

Trust is inherently asymmetrical

In Secure state, the processor:

- Can access both the Secure memory address space and the Non-secure memory address space.

Trustlets can access normal-world memory

- Dubbed “boomerang attack”, Aravind Machiry et al.
- QSEE privilege escalation vulnerability and exploit (CVE-2015-6639, Gal Beniami)
- Userland -> system service (media server) -> … -> QSEE (Widevine trustlet) -> Linux kernel

Shared Nwd and Swd buffers are allocated in Nwd and the (physical) address is mapped into a trustlet virtual address by TEE OS system call from within the trustlet.

Secure Monitor Call instruction from Nwd kernel

```
static noinline int smc_send(uint32_t cmd, phys_addr_t cmd_addr,
                             uint32_t cmd_type, uint8_t wait)
{
    /*tlogd("start to send smc to secure\n");*/
    register u64 x0 asm("x0") = cmd;
    register u64 x1 asm("x1") = cmd_addr;
    register u64 x2 asm("x2") = cmd_type;
    register u64 x3 asm("x3") = cmd_addr >> 32;
    do {
        asm volatile(
                __asmeq("%0", "x0")
                __asmeq("%1", "x0")
                __asmeq("%2", "x1")
                __asmeq("%3", "x2")
                __asmeq("%4", "x3")
                "smc    #0\n"
                : "+r" (x0)
                : "r" (x0), "r" (x1), "r" (x2), "r" (x3));
    } while (x0 == TSP_REQUEST && wait);
    return x0;
}
```

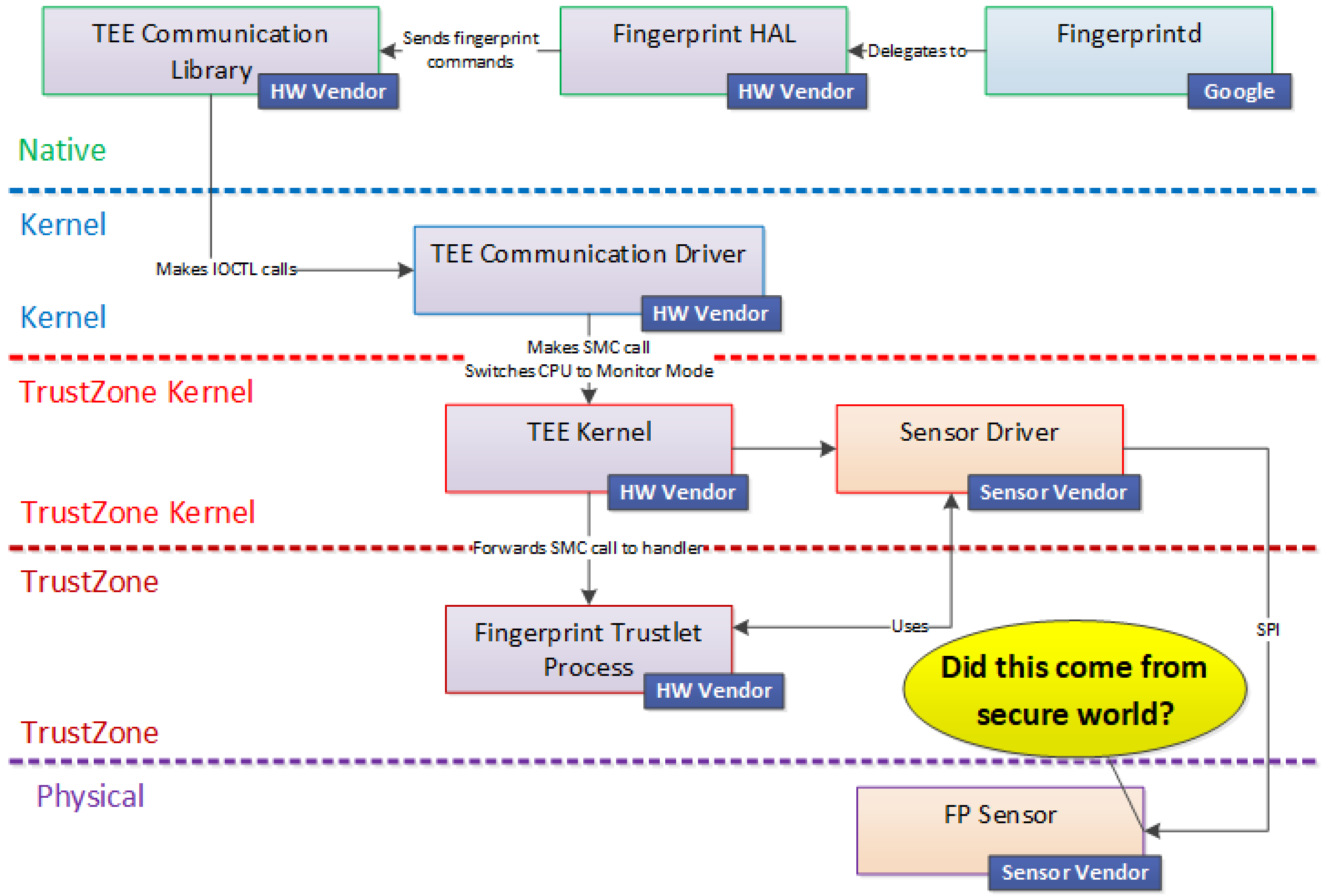
kernel/drivers/hisi/tzdriver/smc.c
Huawei Mate 10 (Kirin 970)

Transfer control to TEE kernel

AxPROT signal

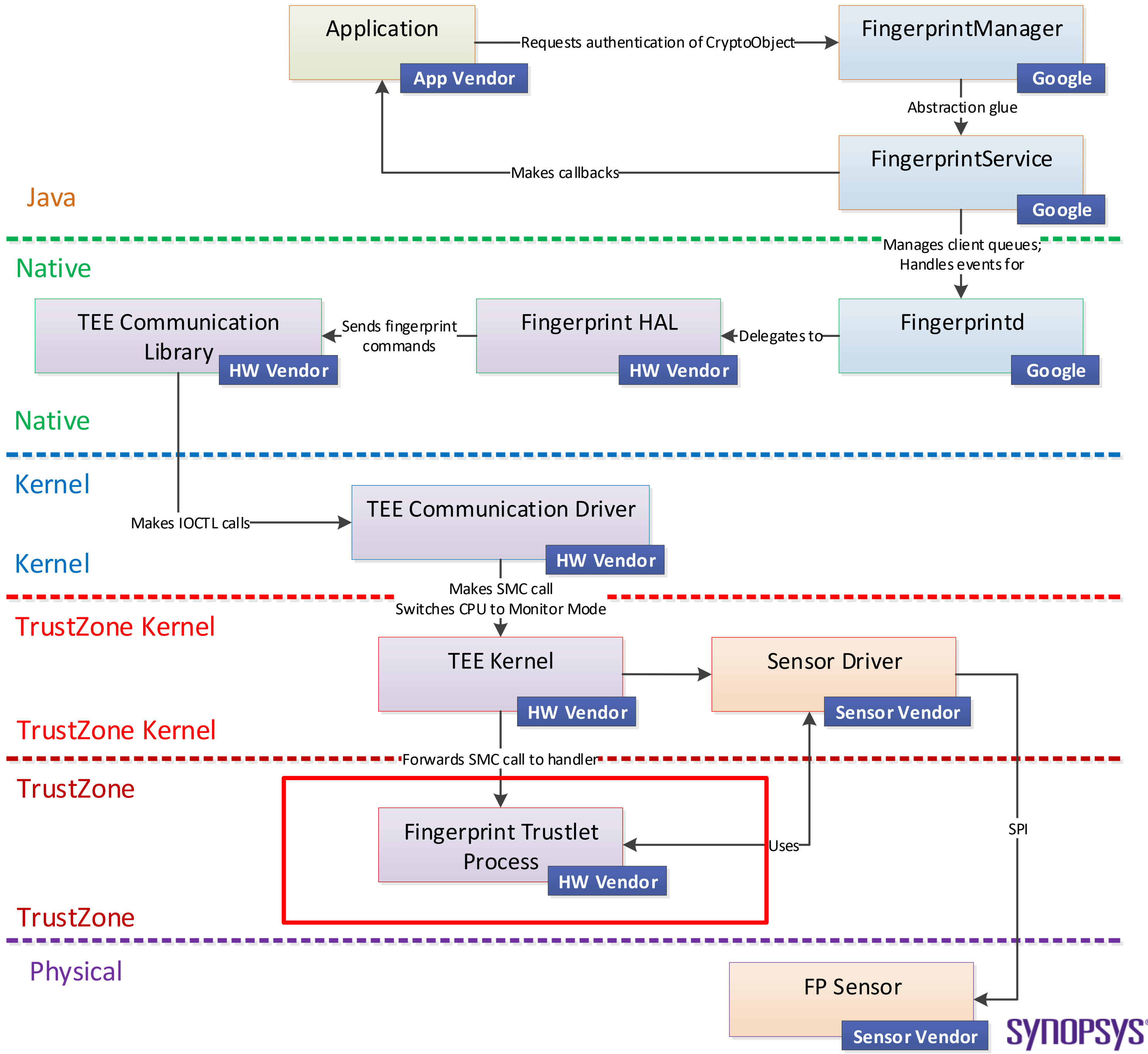
Allows TEE to assert that the signal originated from code that is

- Privileged (as opposed to normal)
- Secure world (as opposed to non-secure world)



Fingerprint Trusted Application

- Swd counterpart of HAL
- Registers handler for SMC calls
- Event handler entry point
- Initialize the hardware driver
- Perform scan
- Create template
- Verify a reading from sensor



Exploring the system

What could go wrong?

Testing a device

1. Build a reference fingerprint system threat model.

Revise and adapt the model(s) for different implementations.

2. Evaluate a device by verifying that the observed controls, e.g. fingerprint template protection, adhere to the reference threat model.

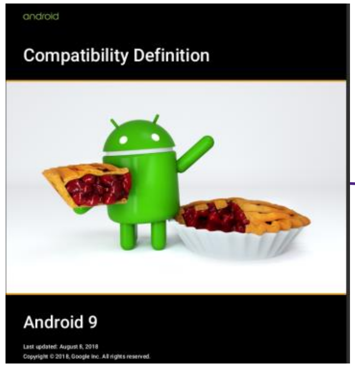
Is the device under test a "valid" implementation of the model?

Looking for evidence of doing stupid things.

Not vulnerability hunting!

Will miss hard to find issues and hard to test issues

Reference model(s)



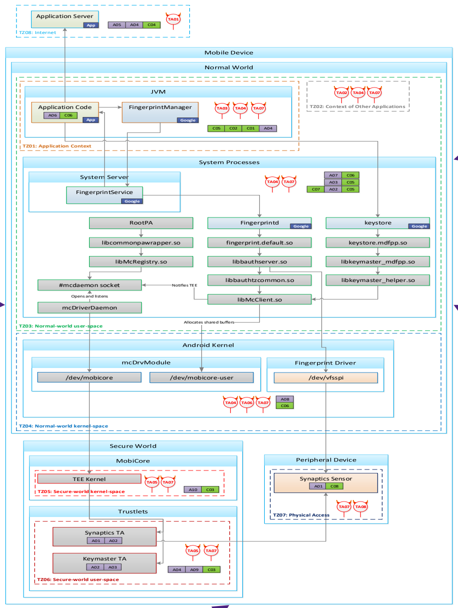
android

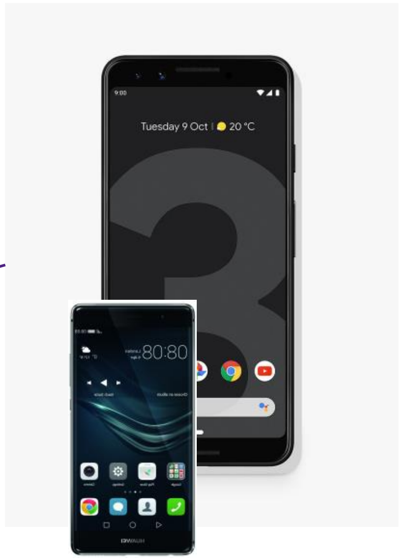
Compatibility Definition

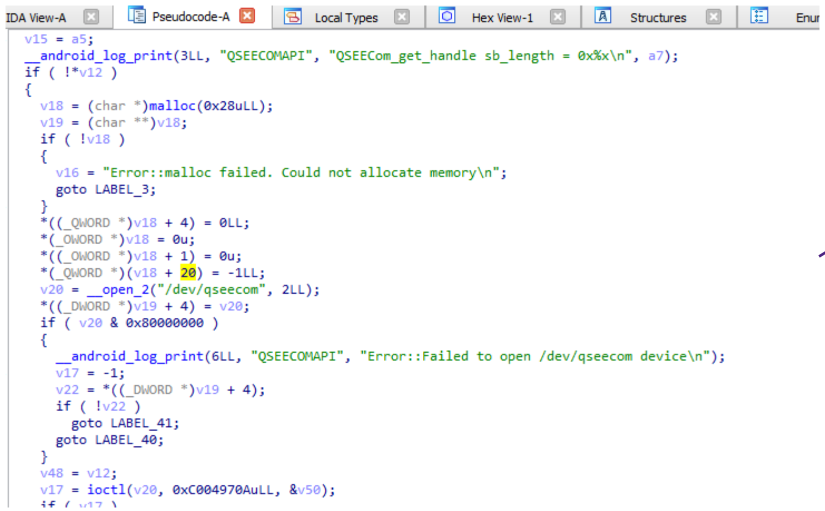
Android 9

Last updated: August 6, 2018

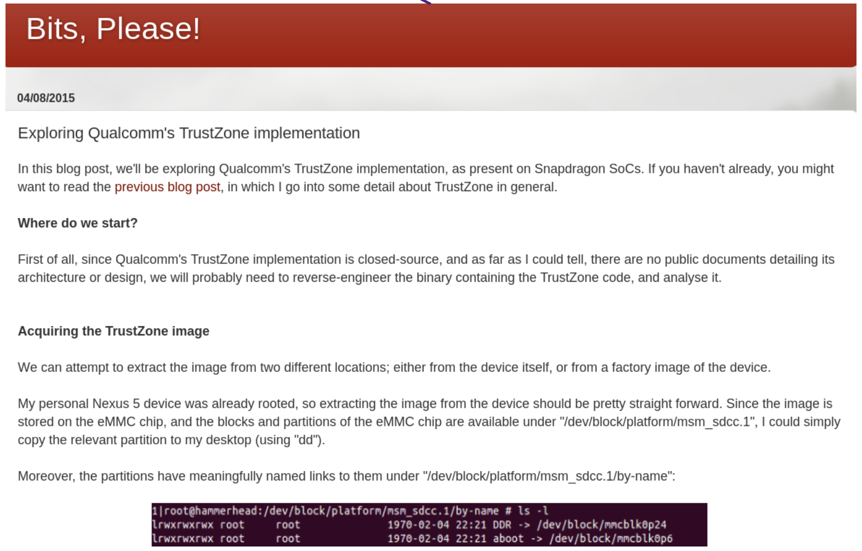
Copyright © 2018, Google Inc. All rights reserved.







```
v15 = a5;
__android_log_print(3LL, "QSEECOMAPI", "QSEECom_get_handle sb_length = 0x%x\n", a7);
if ( !*v12 )
{
  v18 = (char *)malloc(0x28uLL);
  v19 = (char **)v18;
  if ( !v18 )
  {
    v16 = "Error::malloc failed. Could not allocate memory\n";
    goto LABEL_3;
  }
  *((_QWORD *)v18 + 4) = 0LL;
  *(_OWORD *)v18 = 0u;
  *((_OWORD *)v18 + 1) = 0u;
  *(_QWORD *)(v18 + 20) = -1LL;
  v20 = __open_2("/dev/qseecom", 2LL);
  *((_DWORD *)v19 + 4) = v20;
  if ( v20 & 0x80000000 )
  {
    __android_log_print(6LL, "QSEECOMAPI", "Error::Failed to open /dev/qseecom device\n");
    v17 = -1;
    v22 = *((_DWORD *)v19 + 4);
    if ( !v22 )
      goto LABEL_41;
    goto LABEL_40;
  }
  v48 = v12;
  v17 = ioctl(v20, 0xC004970AuLL, &v50);
  if ( v17 )
```



Bits, Please!

04/08/2015

Exploring Qualcomm's TrustZone implementation

In this blog post, we'll be exploring Qualcomm's TrustZone implementation, as present on Snapdragon SoCs. If you haven't already, you might want to read the previous blog post, in which I go into some detail about TrustZone in general.

Where do we start?

First of all, since Qualcomm's TrustZone implementation is closed-source, and as far as I could tell, there are no public documents detailing its architecture or design, we will probably need to reverse-engineer the binary containing the TrustZone code, and analyse it.

Acquiring the TrustZone image

We can attempt to extract the image from two different locations; either from the device itself, or from a factory image of the device.

My personal Nexus 5 device was already rooted, so extracting the image from the device should be pretty straight forward. Since the image is stored on the eMMC chip, and the blocks and partitions of the eMMC chip are available under "/dev/block/platform/msm_sdcc.1", I could simply copy the relevant partition to my desktop (using "dd").

Moreover, the partitions have meaningfully named links to them under "/dev/block/platform/msm_sdcc.1/by-name":

```
1|root@hammerhead:/dev/block/platform/msm_sdcc.1/by-name # ls -l
lrwxrwxrwx root     root              1970-02-04 22:21 DDR -> /dev/block/mmcblk0p24
lrwxrwxrwx root     root              1970-02-04 22:21 aboot -> /dev/block/mmcblk0p6
```

Threat model

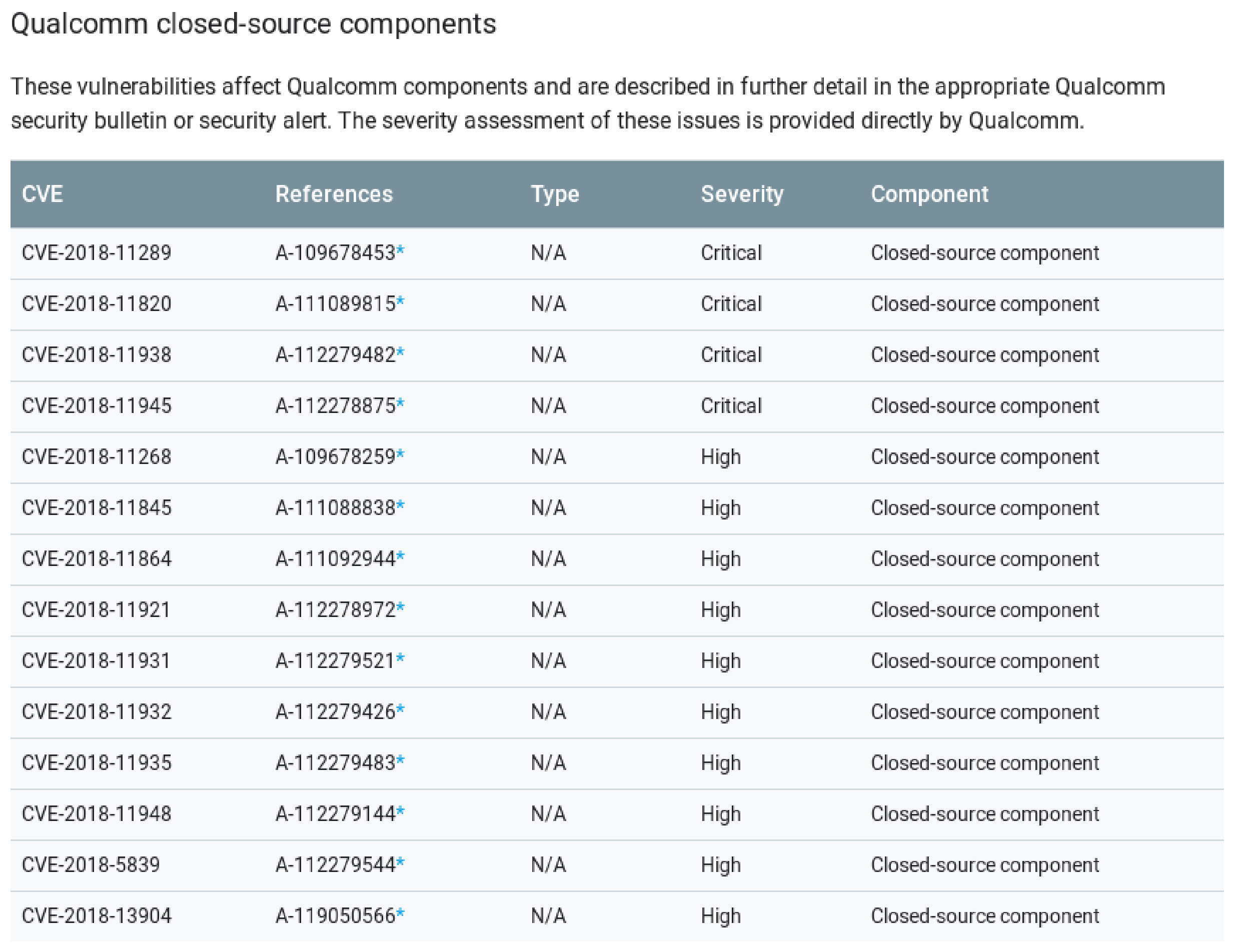
- **Example malicious actors:**
- Privileged malware
- Malicious vendor or careless vendor
- Physical attacker (with access to fingerprint models)
- Rogue trustlet

- **Example controls:**
- Trusted applications are signed
- Template operations inside TEE
- Templates are cryptographically authenticated
- Device specific hardware key
- Virtual device Linux file permissions, SELinux permissions set appropriately

What could go wrong?

- **Many mistakes possible**:

- Weak device configuration
- Sensor accessible from Nwd
- Vulnerabilities in TEEOS
- Vulnerabilities in trusted application
- Template operations in Nwd
- Backdoors
- Template data outside the TEE
- Template data usable on another device
- Scan data outside the TEE
- Template data not authenticated
- Trusted application not signed
- Downgrade attack
- Fingerprint spoofing
- Application errors



Qualcomm closed-source components

These vulnerabilities affect Qualcomm components and are described in further detail in the appropriate Qualcomm security bulletin or security alert. The severity assessment of these issues is provided directly by Qualcomm.

CVE	References	Type	Severity	Component
CVE-2018-11289	A-109678453*	N/A	Critical	Closed-source component
CVE-2018-11820	A-111089815*	N/A	Critical	Closed-source component
CVE-2018-11938	A-112279482*	N/A	Critical	Closed-source component
CVE-2018-11945	A-112278875*	N/A	Critical	Closed-source component
CVE-2018-11268	A-109678259*	N/A	High	Closed-source component
CVE-2018-11845	A-111088838*	N/A	High	Closed-source component
CVE-2018-11864	A-111092944*	N/A	High	Closed-source component
CVE-2018-11921	A-112278972*	N/A	High	Closed-source component
CVE-2018-11931	A-112279521*	N/A	High	Closed-source component
CVE-2018-11932	A-112279426*	N/A	High	Closed-source component
CVE-2018-11935	A-112279483*	N/A	High	Closed-source component
CVE-2018-11948	A-112279144*	N/A	High	Closed-source component
CVE-2018-5839	A-112279544*	N/A	High	Closed-source component
CVE-2018-13904	A-119050566*	N/A	High	Closed-source component

Access the sensor directly from normal world

https://source.android.com/security/authentication/fingerprint-hal

Implementation guidelines

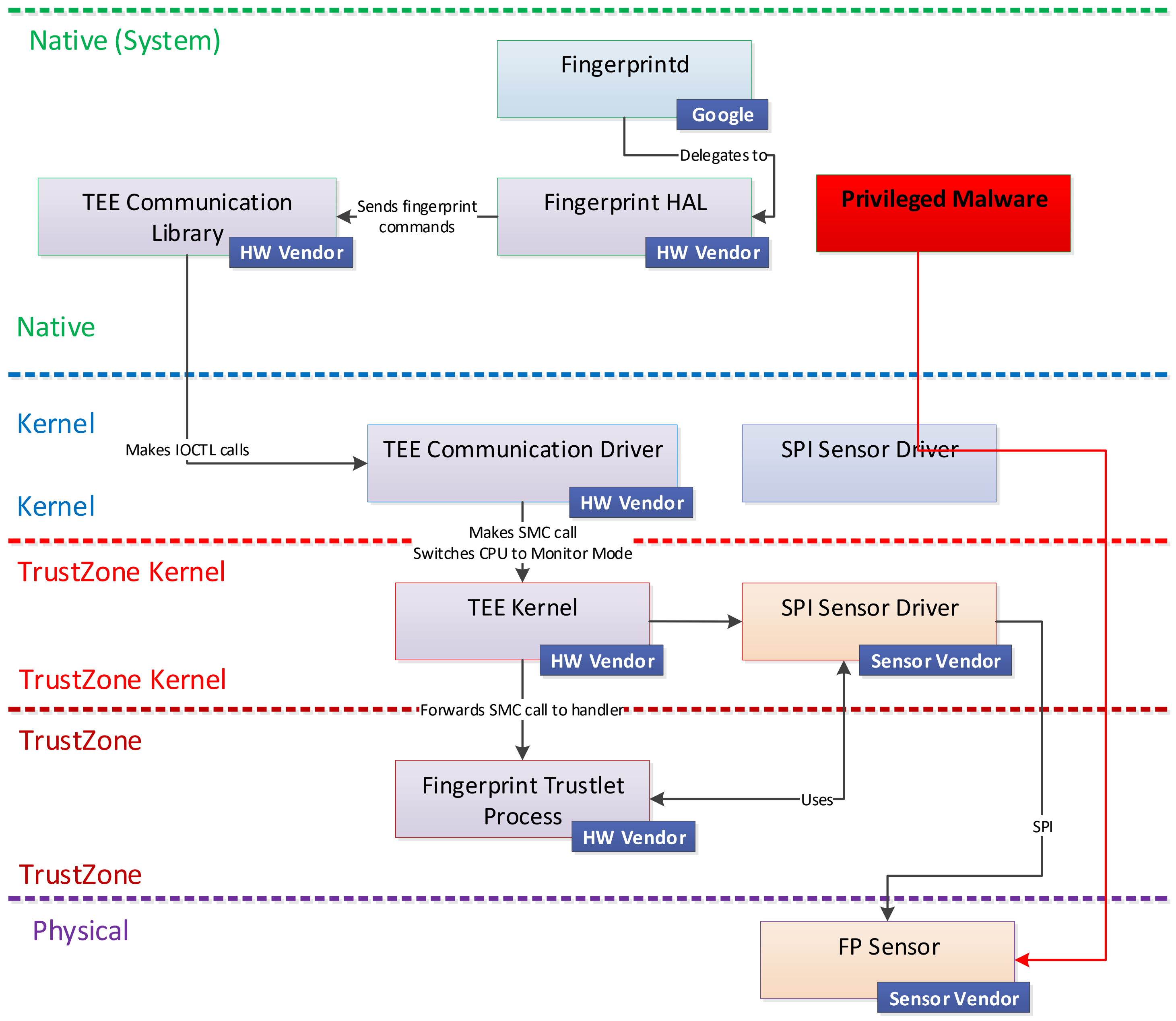
The following Fingerprint HAL guidelines are designed to ensure that fingerprint data is not leaked and is removed when a user is removed from a device:

1. Raw fingerprint data or derivatives (e.g. templates) must never be accessible from outside the sensor driver or TEE. If the hardware supports it, hardware access must be limited to the TEE and protected by an SELinux policy. The Serial Peripheral Interface (SPI) channel must be accessible only to the TEE and there must be an explicit SELinux policy on all device files.

AxPROT signal

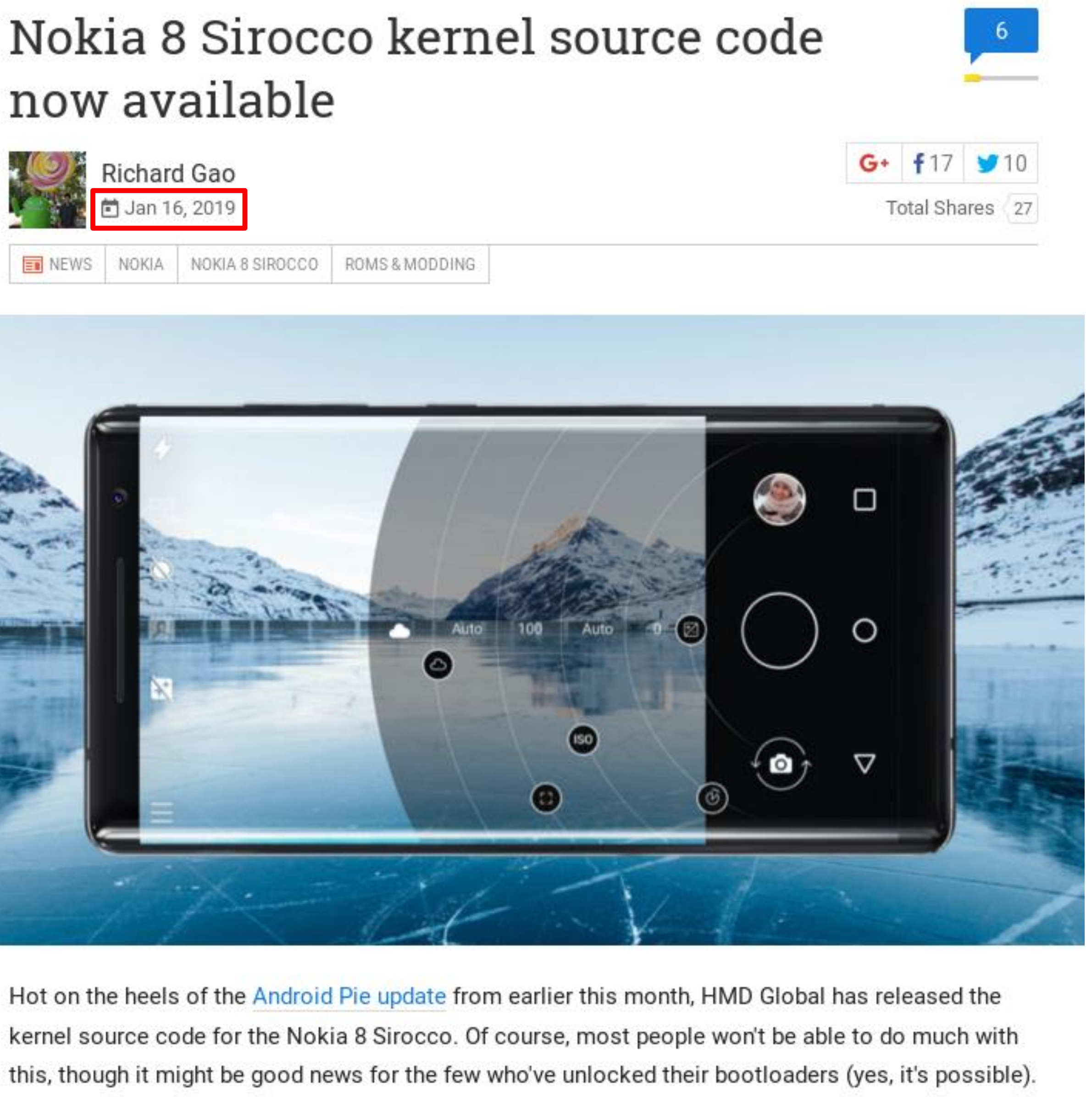
Supposed to checked by fingerprint sensor

- Blackhat USA 15: Fingerprints On Mobile Devices, Tao Wei and Yulong Zhang
- HTC One Max and Samsung Galaxy S5
- Still happens today?



Difficult to test

Source code may be released months later



Nokia 8 Sirocco kernel source code now available

Richard Gao
Jan 16, 2019

NEWS | NOKIA | NOKIA 8 SIROCCO | ROMS & MODDING

Hot on the heels of the Android Pie update from earlier this month, HMD Global has released the kernel source code for the Nokia 8 Sirocco. Of course, most people won't be able to do much with this, though it might be good news for the few who've unlocked their bootloaders (yes, it's possible).

Nokia 8 release date: mid 2018

Disclaimer: not suggesting **this** device vulnerable

Access the kernel device from unprivileged process?

https://source.android.com/security/authentication/fingerprint-hal

Implementation guidelines

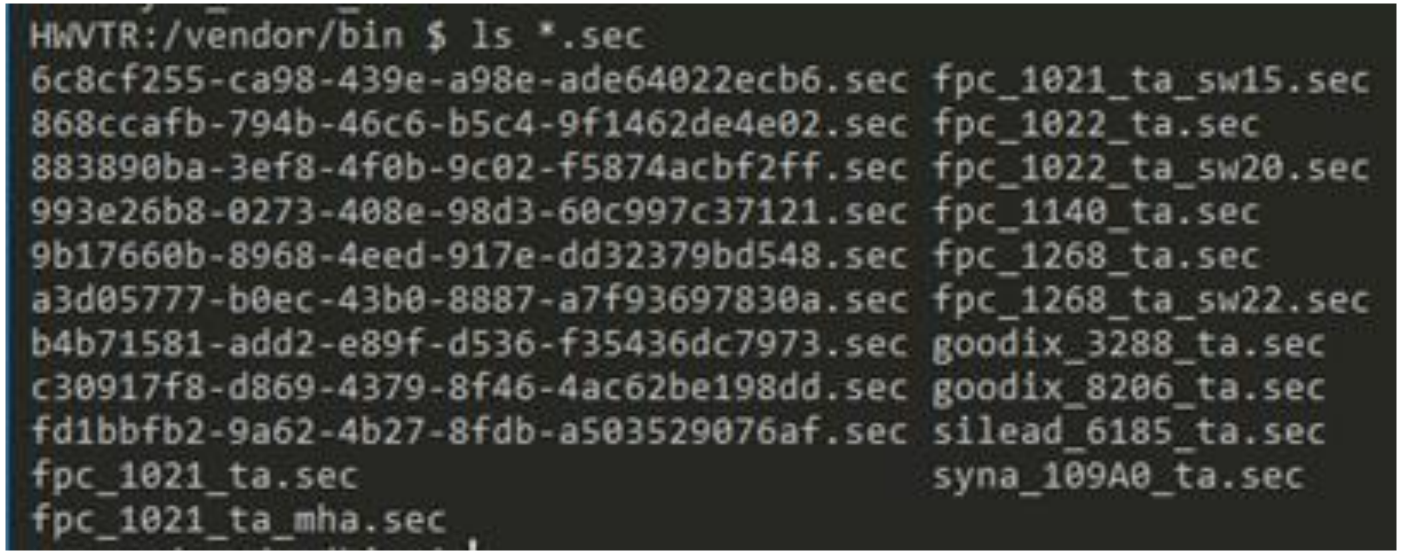
1. … and there must be an explicit SELinux policy on all device files.

We found explicit SELinux policy on all devices though some have weak Linux file permissions

```
judyln:/system $ ls -lZ /dev/goodix_fp
crw-rw-rw- 1 root u:object_r:goodixfingerprintd_device:s0 225,   0 2017-03-26 23:36 /dev/goodix_fp
```

Trustlet signing

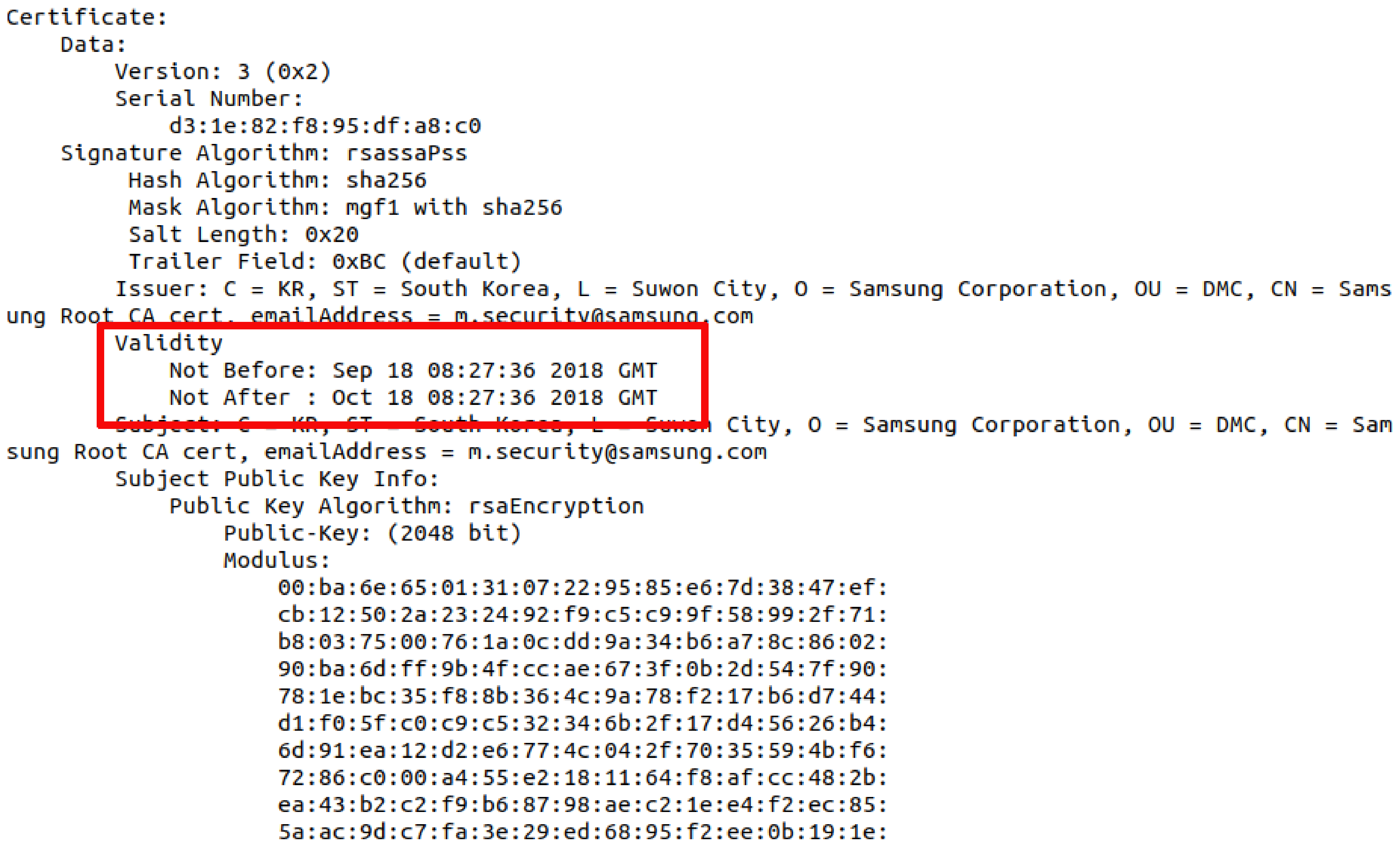
- Qualcomm & Mobicore/Trustonic trustlets are signed
- Huawei trustlets are encrypted – could not verify



```
HWVTR:/vendor/bin $ ls *.sec
6c8cf255-ca98-439e-a98e-ade64022ecb6.sec fpc_1021_ta_sw15.sec
868ccafb-794b-46c6-b5c4-9f1462de4e02.sec fpc_1022_ta.sec
883890ba-3ef8-4f0b-9c02-f5874acbf2ff.sec fpc_1022_ta_sw20.sec
993e26b8-0273-408e-98d3-60c997c37121.sec fpc_1140_ta.sec
9b17660b-8968-4eed-917e-dd32379bd548.sec fpc_1268_ta.sec
a3d05777-b0ec-43b0-8887-a7f93697830a.sec fpc_1268_ta_sw22.sec
b4b71581-add2-e89f-d536-f35436dc7973.sec goodix_3288_ta.sec
c30917f8-d869-4379-8f46-4ac62be198dd.sec goodix_8206_ta.sec
fd1bbfb2-9a62-4b27-8fdb-a503529076af.sec silead_6185_ta.sec
fpc_1021_ta.sec                          syna_109A0_ta.sec
fpc_1021_ta_mha.sec
```

Encrypted trustlets on Huawei P10 VTR-AL00 device

Signature chain sometimes expired



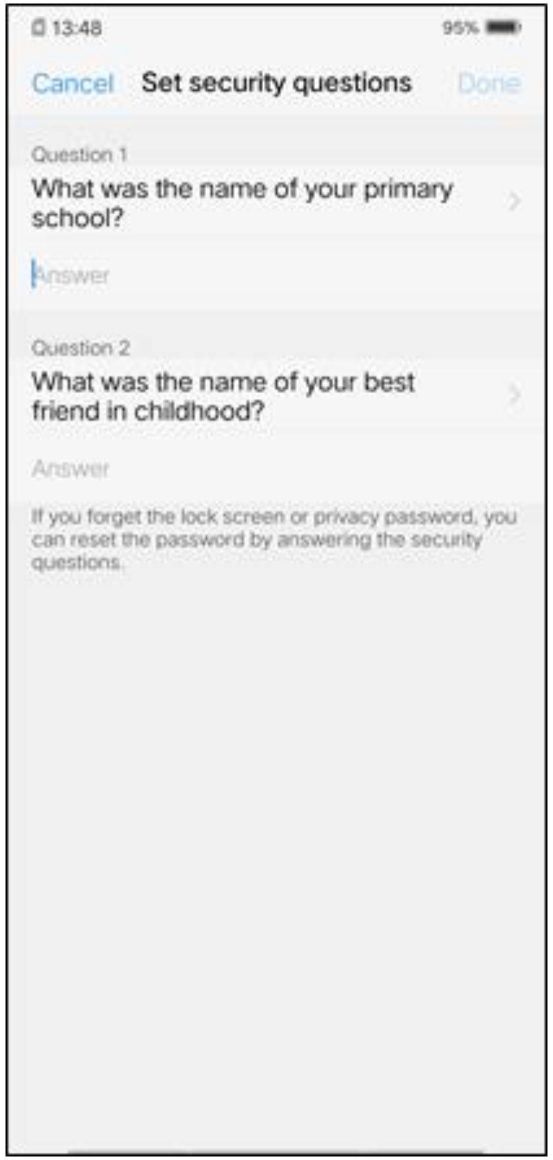
```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            d3:1e:82:f8:95:df:a8:c0
    Signature Algorithm: rsassaPss
         Hash Algorithm: sha256
         Mask Algorithm: mgf1 with sha256
         Salt Length: 0x20
         Trailer Field: 0xBC (default)
        Issuer: C = KR, ST = South Korea, L = Suwon City, O = Samsung Corporation, OU = DMC, CN = Sams
ung Root CA cert, emailAddress = m.security@samsung.com
        Validity
            Not Before: Sep 18 08:27:36 2018 GMT
            Not After : Oct 18 08:27:36 2018 GMT
        Subject: C = KR, ST = South Korea, L = Suwon City, O = Samsung Corporation, OU = DMC, CN = Sam
sung Root CA cert, emailAddress = m.security@samsung.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:ba:6e:65:01:31:07:22:95:85:e6:7d:38:47:ef:
                    cb:12:50:2a:23:24:92:f9:c5:c9:9f:58:99:2f:71:
                    b8:03:75:00:76:1a:0c:dd:9a:34:b6:a7:8c:86:02:
                    90:ba:6d:ff:9b:4f:cc:ae:67:3f:0b:2d:54:7f:90:
                    78:1e:bc:35:f8:8b:36:4c:9a:78:f2:17:b6:d7:44:
                    d1:f0:5f:c0:c9:c5:32:34:6b:2f:17:d4:56:26:b4:
                    6d:91:ea:12:d2:e6:77:4c:04:2f:70:35:59:4b:f6:
                    72:86:c0:00:a4:55:e2:18:11:64:f8:af:cc:48:2b:
                    ea:43:b2:c2:f9:b6:87:98:ae:c2:1e:e4:f2:ec:85:
                    5a:ac:9d:c7:fa:3e:29:ed:68:95:f2:ee:0b:19:1e:
```

Chain of trust

[C-1-8] First establish chain of trust (PIN/pattern/password)

All devices enforced this …

…, however, weak stock questions allow PIN reset


13:48 95%

Cancel Set security questions Done

Question 1

What was the name of your primary school?

Answer

Question 2

What was the name of your best friend in childhood?

Answer

If you forget the lock screen or privacy password, you can reset the password by answering the security questions.

Fingerprint spoofing

[C-1-5] MUST rate limit attempts for at least 30 seconds after five false trials for fingerprint verification.

All devices locked the screen after five attempts

… but then some continued to process fingerprint attempts

… and logged the result along with quality metrics

Attack: copy fingerprint template from another device

Implementation guidelines:
“Fingerprint templates must be signed with a private, device-specific key.”

Defense: derive a device specific key

```
__int64 __cdecl fp_get_uniquekey()
{
    int(*v0)(int, const char *, const char *, ...); // x19
    unsigned int v1; // w0
    __int64(*v2)(__int64, const char *, ...); // x3

    v0 = g_pLogFunction;
    (*g_pLogFunction)(8LL, "%s Init", qword_6A3320);
    (*v0)(8LL, "fp_get_uniquekey start");
    v1 = qsee_kdf(0, 32, aSamsungSecurit, 32, aFpDeviceKey, 32, g_UniqueKey, 64);
    v2 = *v0;
    if (!v1)
        return v2(8LL, "fp_get_uniquekey success end");
    v2(8LL, "qsee_kdf error rv = %d", v1);
    return (*v0)(8LL, "get_uniquekey failed");
}
```

Template wrapping (encrypt and HMAC) is done in the trustlet

Attack: exploit vulnerabilities in trusted application

- Written in C
- Buffer overflows, integer wrapping, out of secure world writes

- Complex command/template parsing logic
- Compile-time hardening (e.g. stack protector) mostly **absent**

Demo

What’s next?

Incremental improvements

Improved isolation

- **Trustzone:**
- The two worlds share the same hardware: isolation achieved through the use of CPU registers and a non-secure (NS) bit.
- **Separate processor on same SoC**:
- Qualcomm Snapdragon SDM845 and SDM855 Secure Unit Processor
  - Closer parity with Apple’s Secure Enclave
- Other vendors to follow suit?
  - E.g. google bought HTC Mobile

- Addresses the issue that TrustZone applications now form a large TCB
- Doesn’t solve many of the other design properties (nor does it claim to)

Key attestation

- Attest that key is stored in a device's hardware-backed keystore.
- Relying party checks certificate chain signed by “Google attestation root key”

- Very small step closer to “evidence” for a second factor
- Keymaster 3:

```
enum class HardwareAuthenticatorType : uint32_t
{
    NONE = 0u, // 0
    PASSWORD = 1 << 0,
    FINGERPRINT = 1 << 1,
    ANY = UINT32_MAX,
};
```

https://source.android.com/security/keystore/tags#user_auth_type

https://developer.android.com/training/articles/security-key-attestation

Key takeaways

- There isn't a single "Android" fingerprint system
  - Vendors have considerable flexibility to implement subject to CDD and guidance constraints
- Plenty of opportunities to make mistakes:
  - Nwd Apps
  - Nwd and Swd fingerprint components
- TEE is a high-value target:
  - Image and template processing involves plenty of complex logic: bugs highly likely
  - Lack of defensive measures such as stack canaries being used
- TEE TCB is becoming large.
  - E.g. rogue/vulnerable trusted application could derive template key

A parting question

Can the fingerprint design on Android somehow be used to support two-factor auth?

Thank you!

SYNOPSYS®

Silicon to Software™