Trust Dies in Darkness: Shedding Light on Samsung's TrustZone Cryptographic Design

TEL AVIV UNIVERSITY אוניברסיטת תל אביב

Alon Shakevsky, Eyal Ronen, Avishai Wool
Tel Aviv University

Proprietary OEM implementation + Pitfalls of cryptographic APIs = Severe design flaws

Extended paper: https://eprint.iacr.org/2022/208.pdf

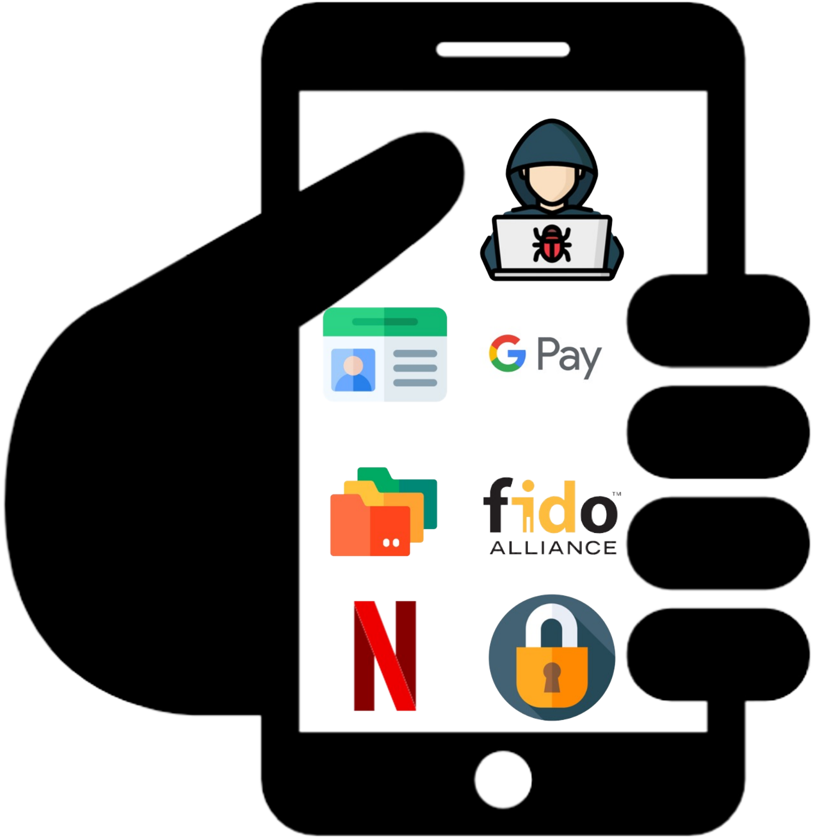
Agenda

- **Introduction** — Background and motivation
- **Keymaster TA Analysis** — Recovering hardware-protected keys
- **Implications** — Breaking higher-level protocols
- **Discussion** — Main takeaways from our research

The need for Trusted Execution Environments (TEEs)

Designed using resources from Flaticon.com

The need for Trusted Execution Environments (TEEs)



Designed using resources from Flaticon.com

Proprietary TrustZone Operating Systems (TZOS)

The implementation of the TrustZone OS is left to vendors.
Samsung devices have one of the following three TZOS:

- Qualcomm Secure Execution Environment (QSEE)
- Kinibi by Trustonic
- TEEGRIS by Samsung

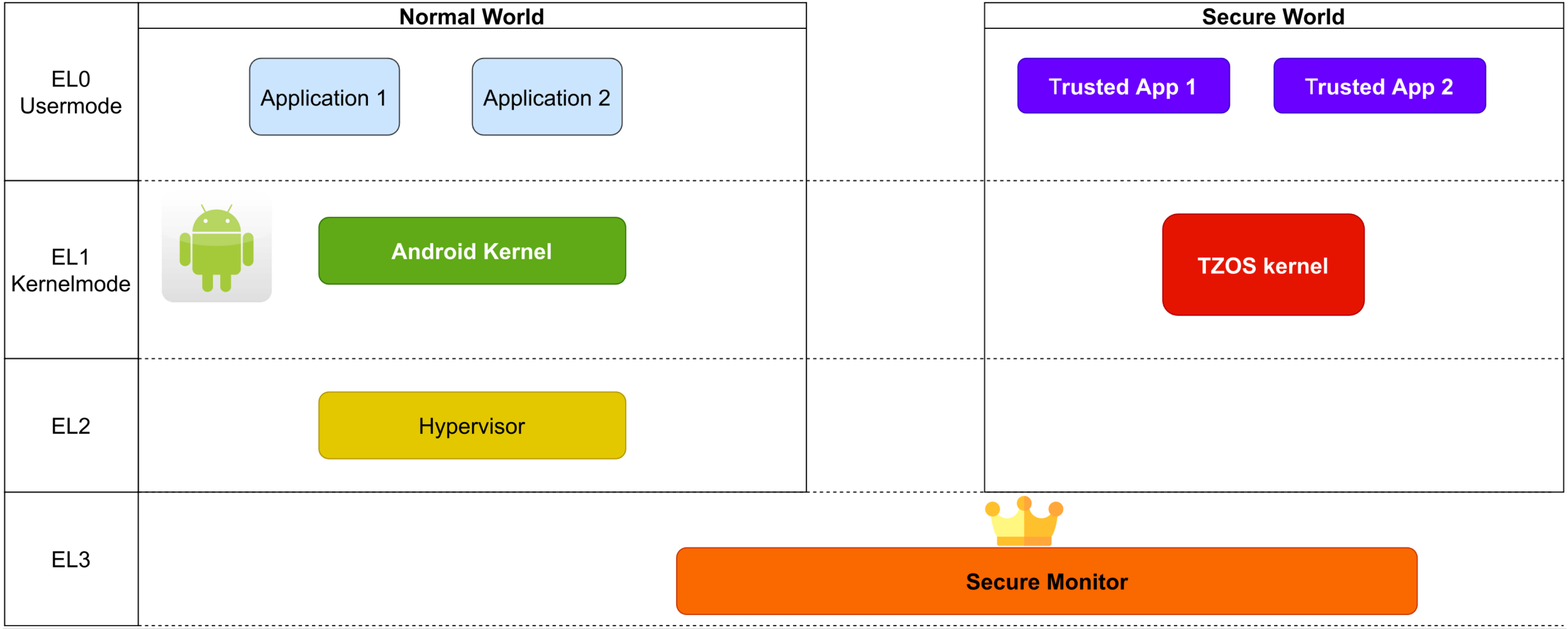
Such vendors maintain secrecy around their implementation and design of TZOSs and TAs.

Research questions

1. Does the hardware-based protection of cryptographic keys remain secure even when the Normal World is compromised?
2. How does the cryptography design of this protection affect the security of various protocols that rely on its security?

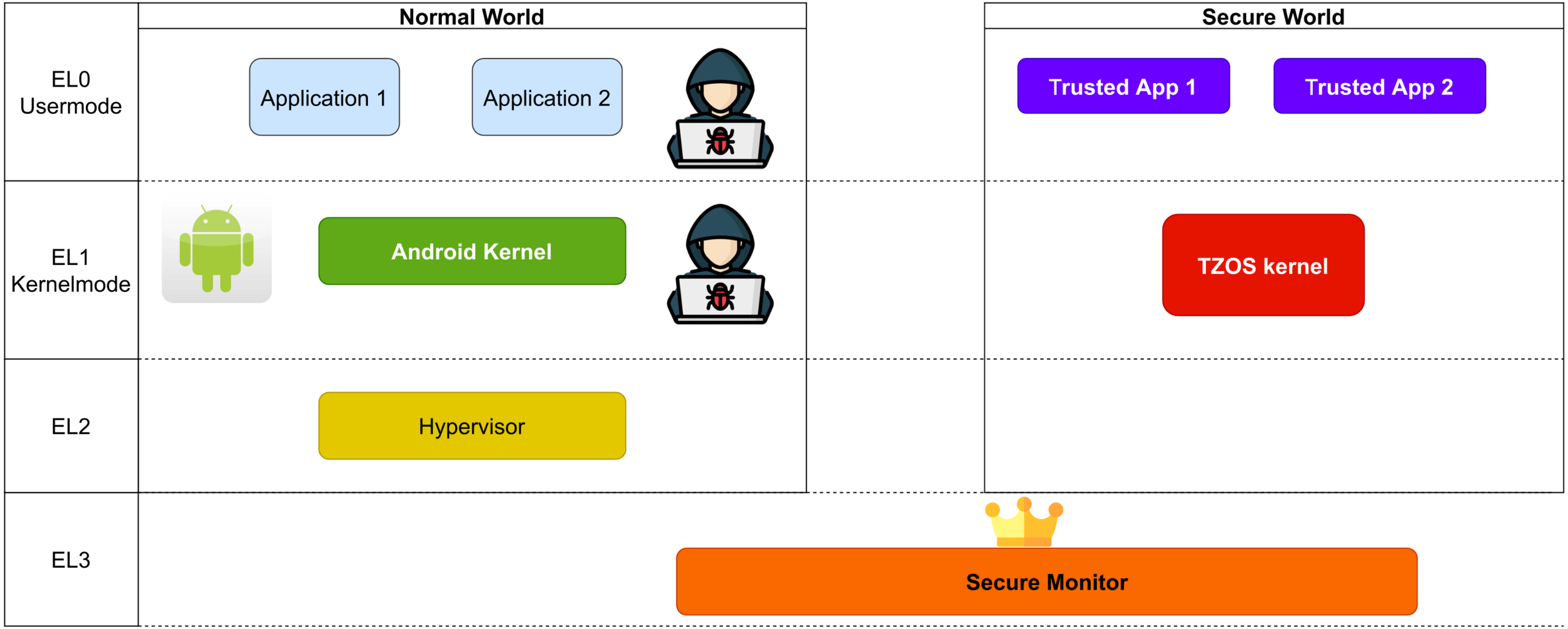
Designed using resources from Flaticon.com

ARM TrustZone - Attack Model



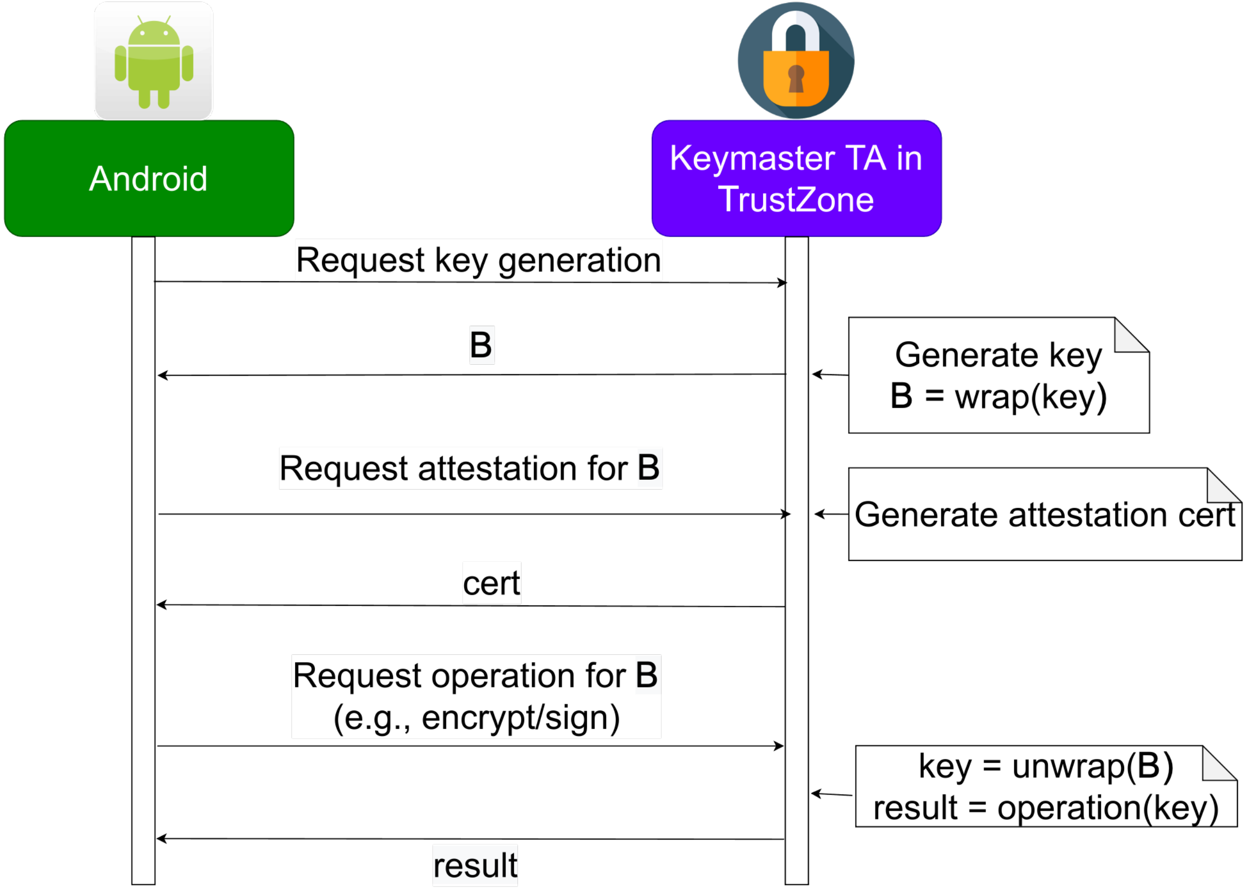
Designed using resources from Flaticon.com

ARM TrustZone - Attack Model

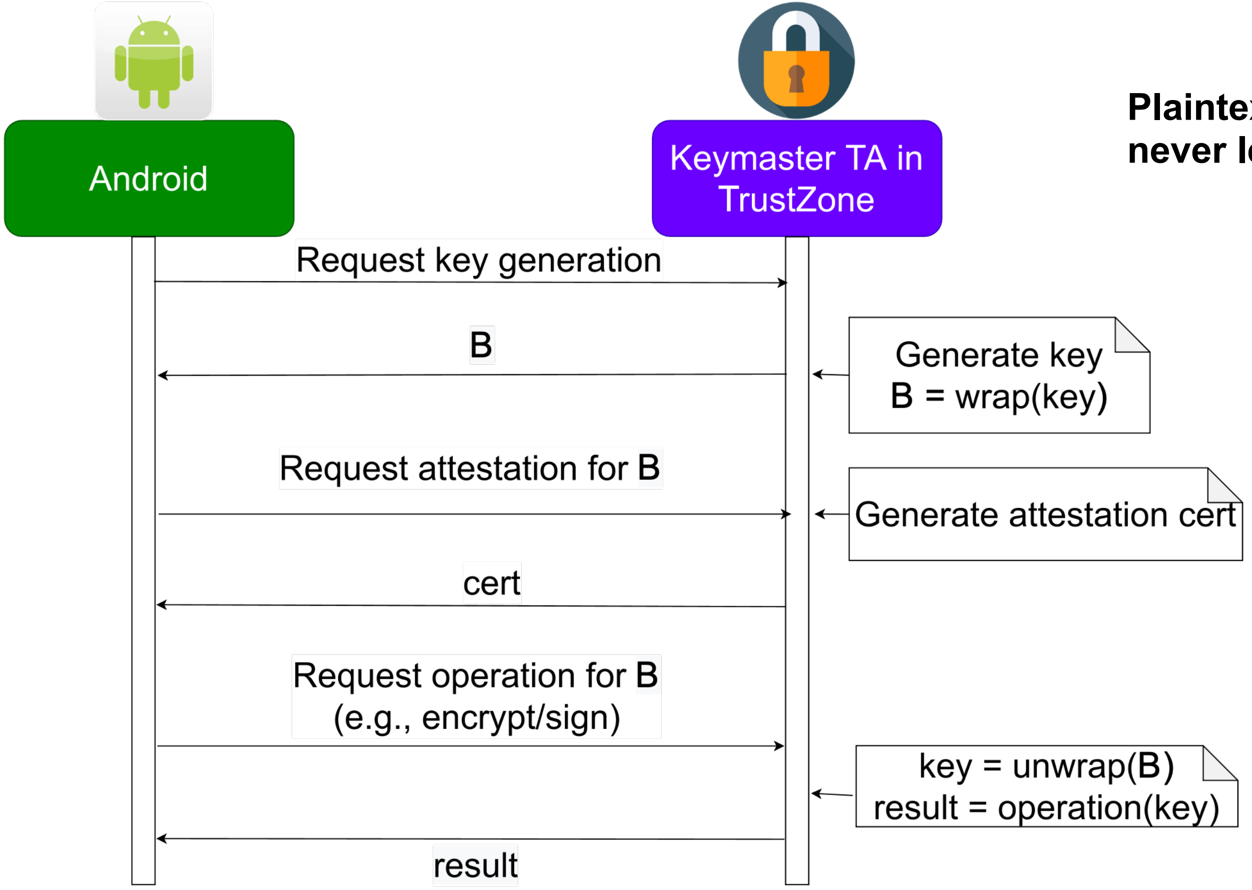


Designed using resources from Flaticon.com

Android Hardware-Backed Keystore flow



Android Hardware-Backed Keystore flow



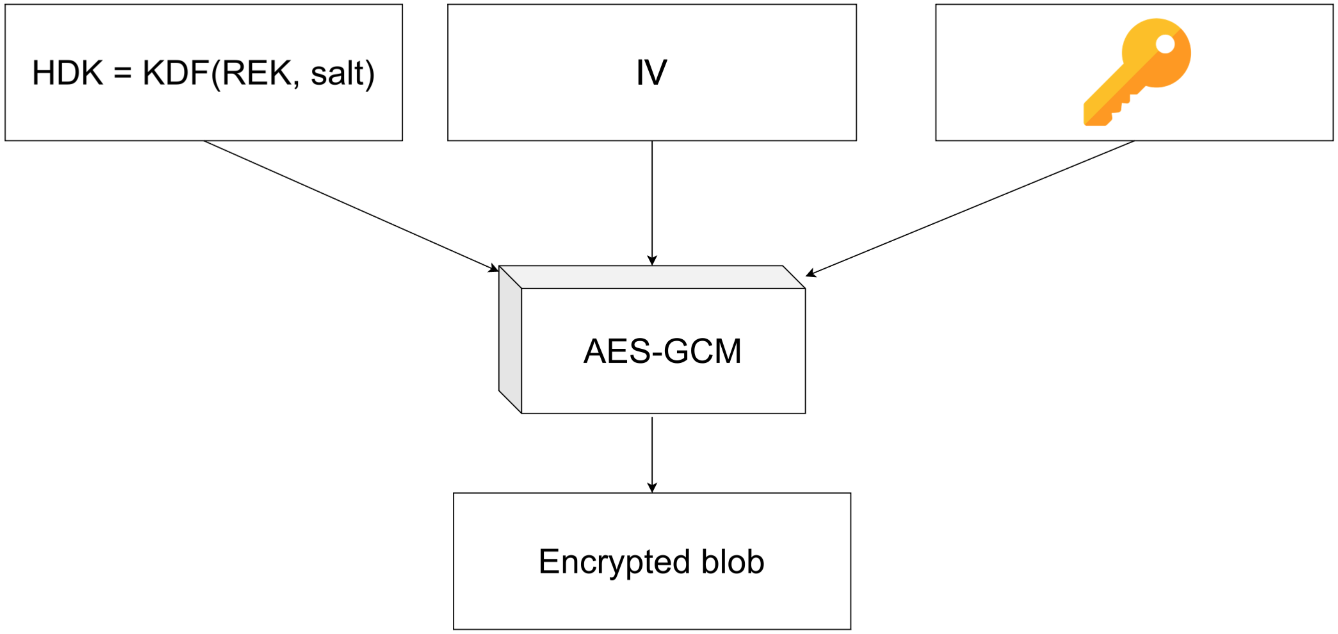
Plaintext key material never leaves the TZOS

Agenda

- **Introduction** — Background and motivation
- **Keymaster TA Analysis** — Recovering hardware-protected keys
- **Implications** — Breaking higher-level protocols
- **Discussion** — Main takeaways from our research

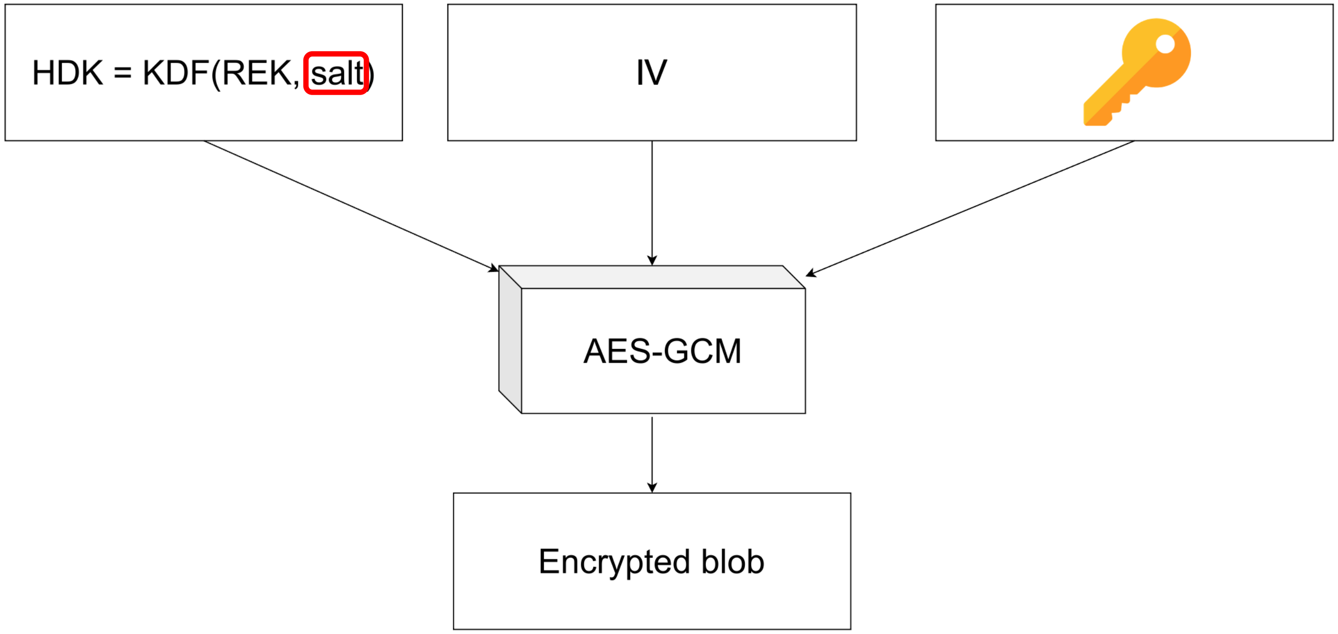
Key Blob Encryption

The Keymaster TA encrypts key material inside blobs.



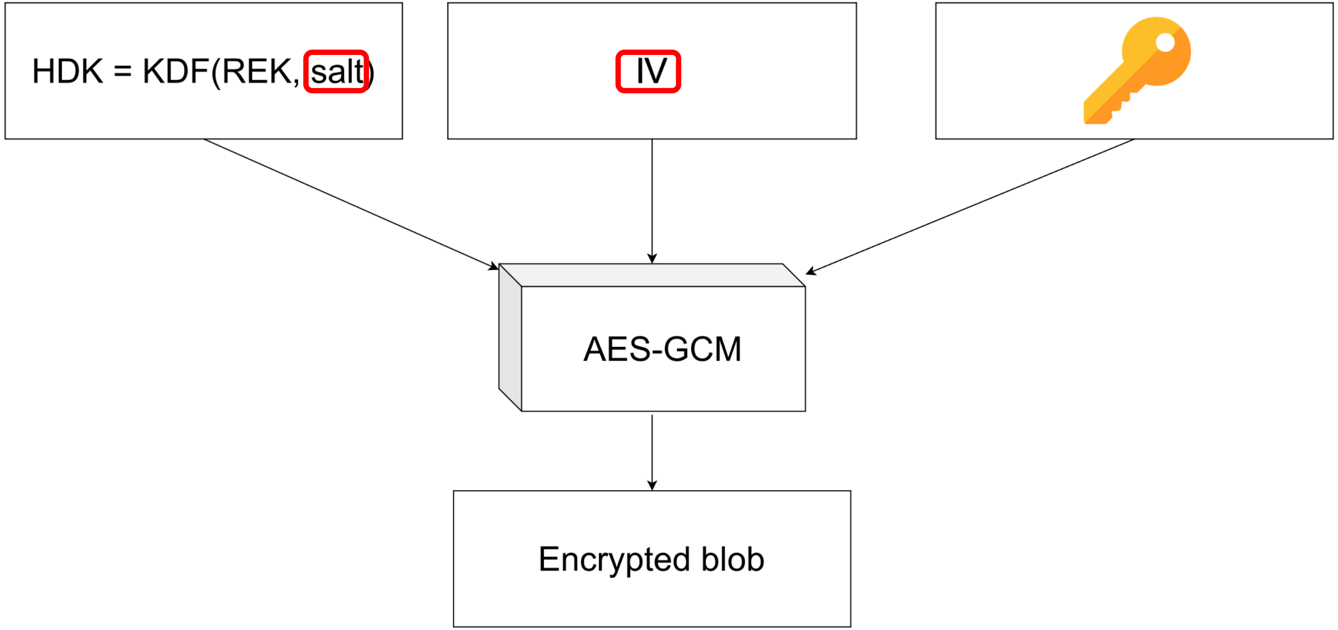
Key Blob Encryption

The Keymaster TA encrypts key material inside blobs.



Key Blob Encryption

The Keymaster TA encrypts key material inside blobs.



KDF versions of key blobs in Samsung devices

Samsung devices are MDFPP CC certified.

The salt value is the SHA256 digest of the concatenation of the values (assuming the Application ID is “id” and the Application Data is “data”):

v15 blob
"MDFPP HW Keymaster HEK v15\x00"
"ID"
"\x02\x00\x00\x00"
"id"
"DATA"
"\x04\x00\x00\x00"
"data"

v20-s9 blob
"MDFPP HW Keymaster HEK v20\x00"
root_of_trust
"ID"
"\x02\x00\x00\x00"
"id"
"DATA"
"\x04\x00\x00\x00"
"data"
integrity_flags

v20-s10 blob
"MDFPP HW Keymaster HEK v20\x00"
root_of_trust
"ID"
"\x02\x00\x00\x00"
"id"
"DATA"
"\x04\x00\x00\x00"
"data"
integrity_flags
hek_randomness

KDF versions of key blobs in Samsung devices

Samsung devices are MDFPP CC certified.
The salt value is the SHA256 digest of the concatenation of the values (assuming the Application ID is “id” and the Application Data is “data”):

v15 blob
"MDFPP HW Keymaster HEK v15\x00"
"ID"
"\x02\x00\x00\x00"
"id"
"DATA"
"\x04\x00\x00\x00"
"data"

v20-s9 blob
"MDFPP HW Keymaster HEK v20\x00"
root_of_trust
"ID"
"\x02\x00\x00\x00"
"id"
"DATA"
"\x04\x00\x00\x00"
"data"
integrity_flags

v20-s10 blob
"MDFPP HW Keymaster HEK v20\x00"
root_of_trust
"ID"
"\x02\x00\x00\x00"
"id"
"DATA"
"\x04\x00\x00\x00"
"data"
integrity_flags
hek_randomness

IV Reuse Attack (v15/v20-s9)

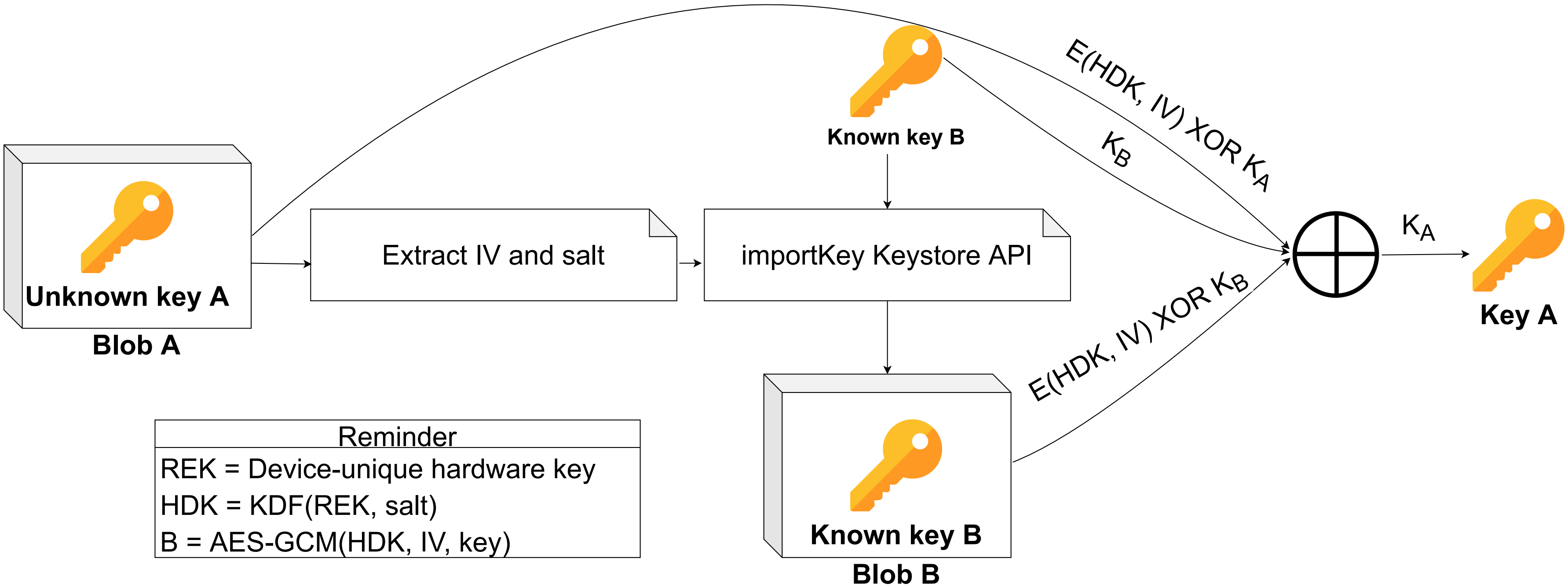
- The Android client can control the salt -> key reuse

IV Reuse Attack (v15/v20-s9)

- The Android client can control the salt -> key reuse
- The Android client can control the IV -> IV reuse
- AES-GCM + key reuse + iv reuse -> decryption

IV Reuse Attack (v15/v20-s9)

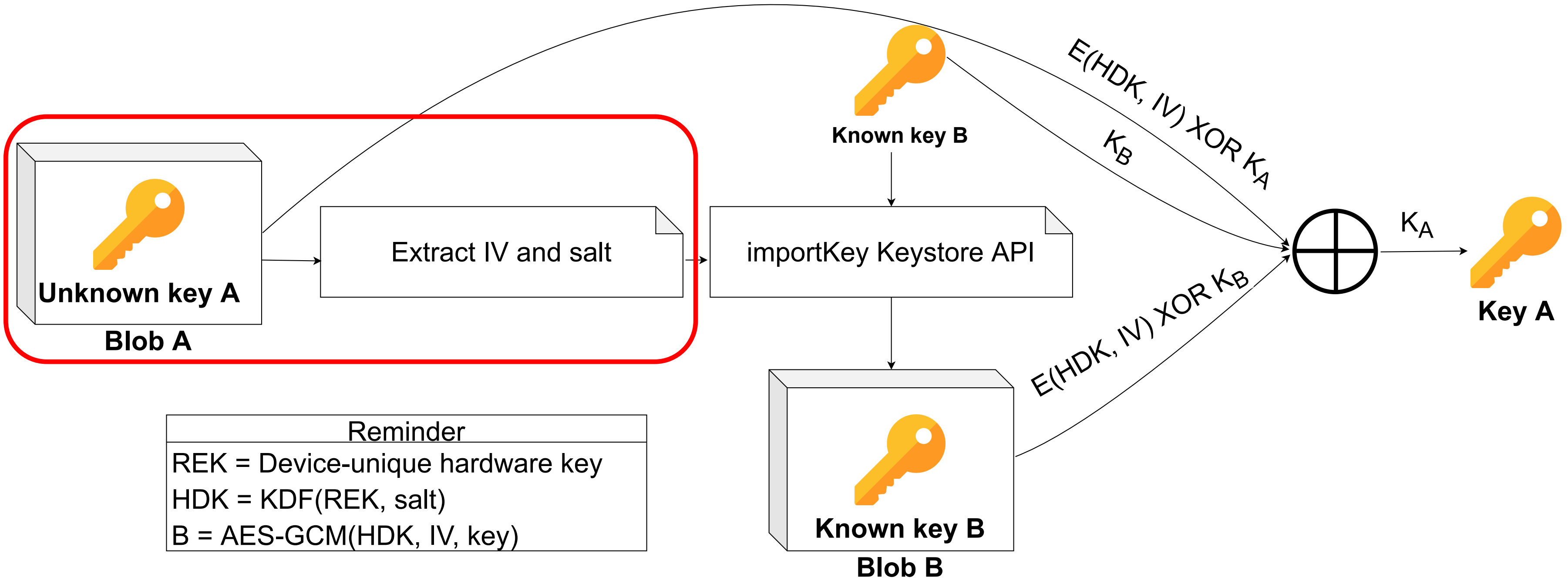
- The Android client can control the salt -> key reuse
- The Android client can control the IV -> IV reuse
- AES-GCM + key reuse + iv reuse -> decryption



Reminder
REK = Device-unique hardware key
HDK = KDF(REK, salt)
B = AES-GCM(HDK, IV, key)

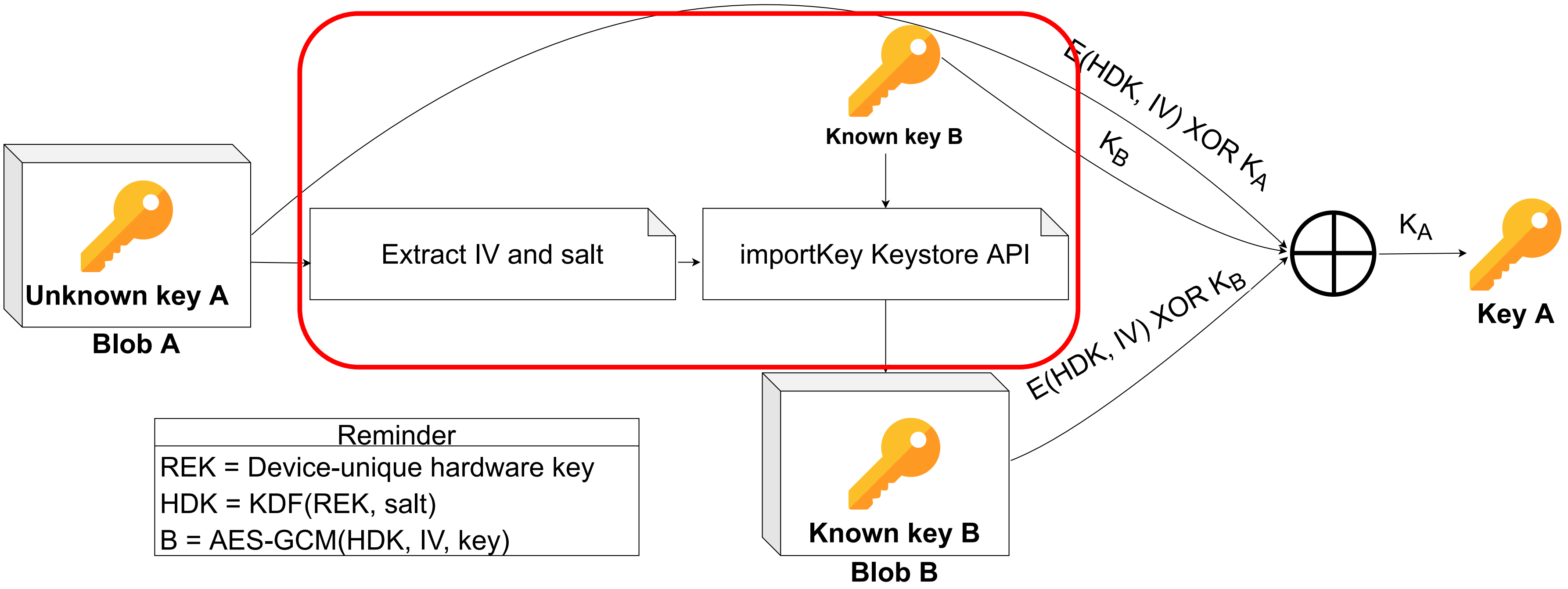
IV Reuse Attack (v15/v20-s9)

- The Android client can control the salt -> key reuse
- The Android client can control the IV -> IV reuse
- AES-GCM + key reuse + iv reuse -> decryption



IV Reuse Attack (v15/v20-s9)

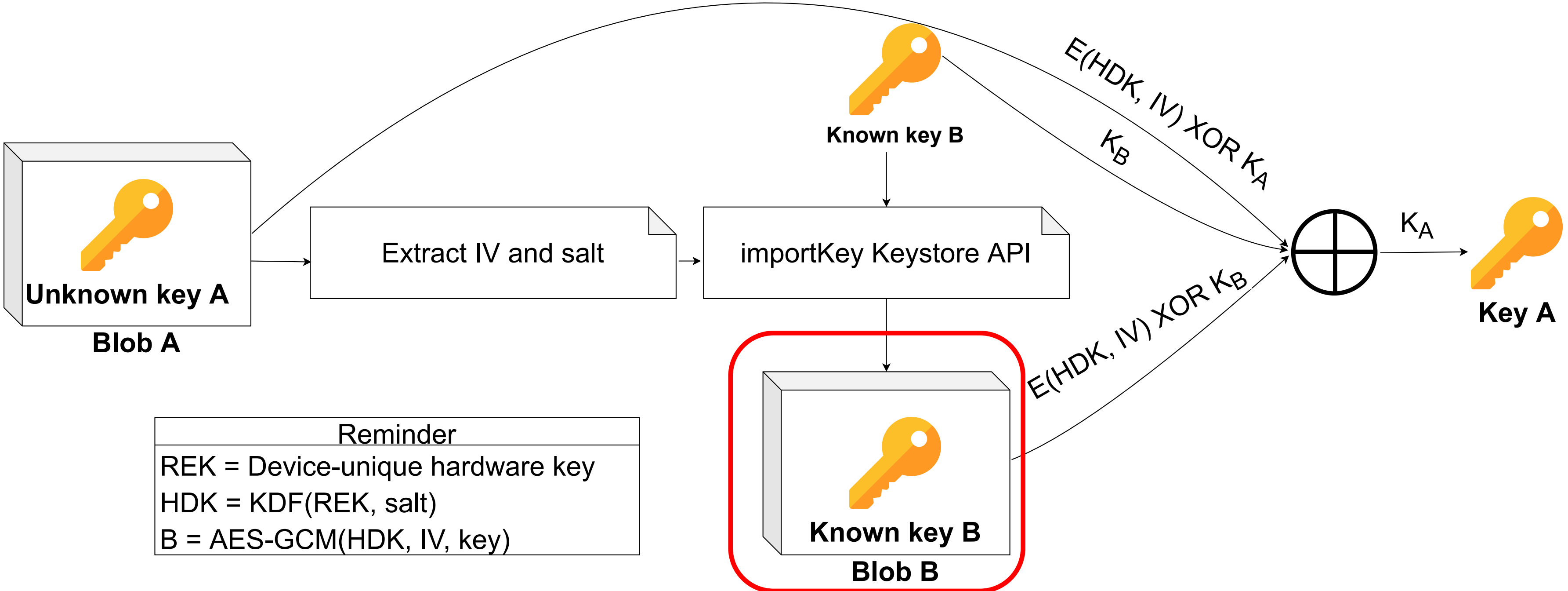
- The Android client can control the salt -> key reuse
- The Android client can control the IV -> IV reuse
- AES-GCM + key reuse + iv reuse -> decryption



Reminder
REK = Device-unique hardware key
HDK = KDF(REK, salt)
B = AES-GCM(HDK, IV, key)

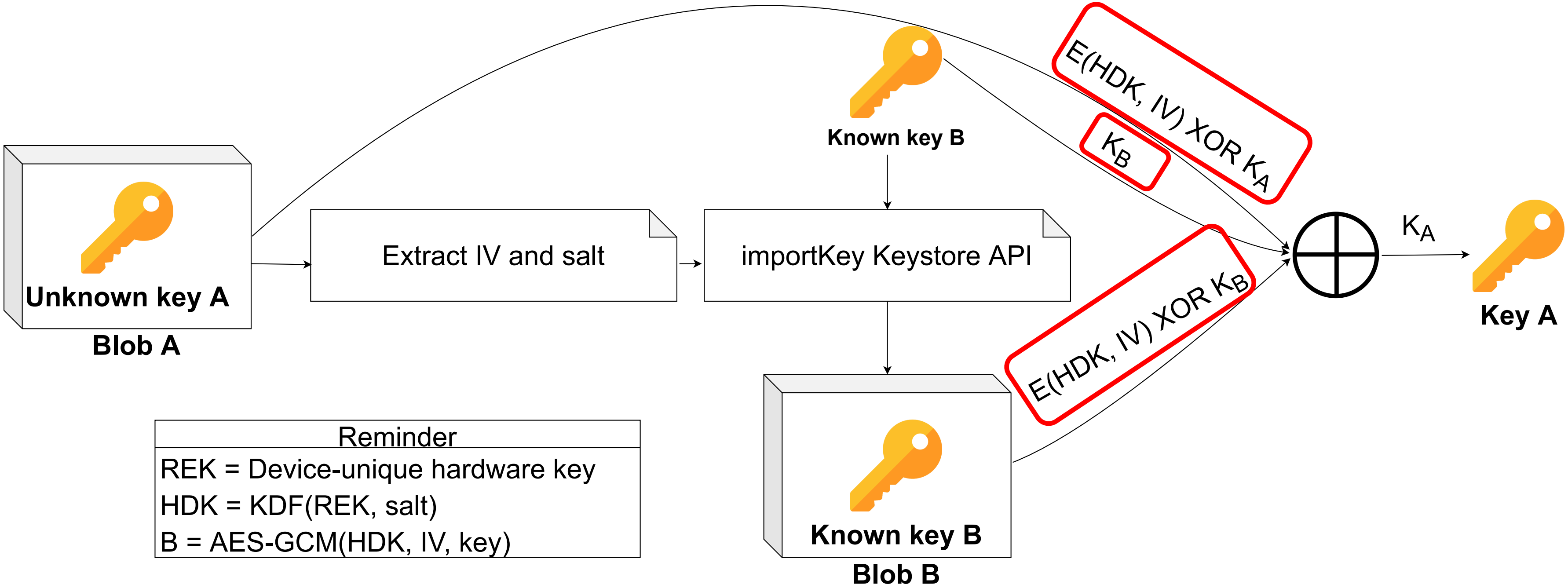
IV Reuse Attack (v15/v20-s9)

- The Android client can control the salt -> key reuse
- The Android client can control the IV -> IV reuse
- AES-GCM + key reuse + iv reuse -> decryption



IV Reuse Attack (v15/v20-s9)

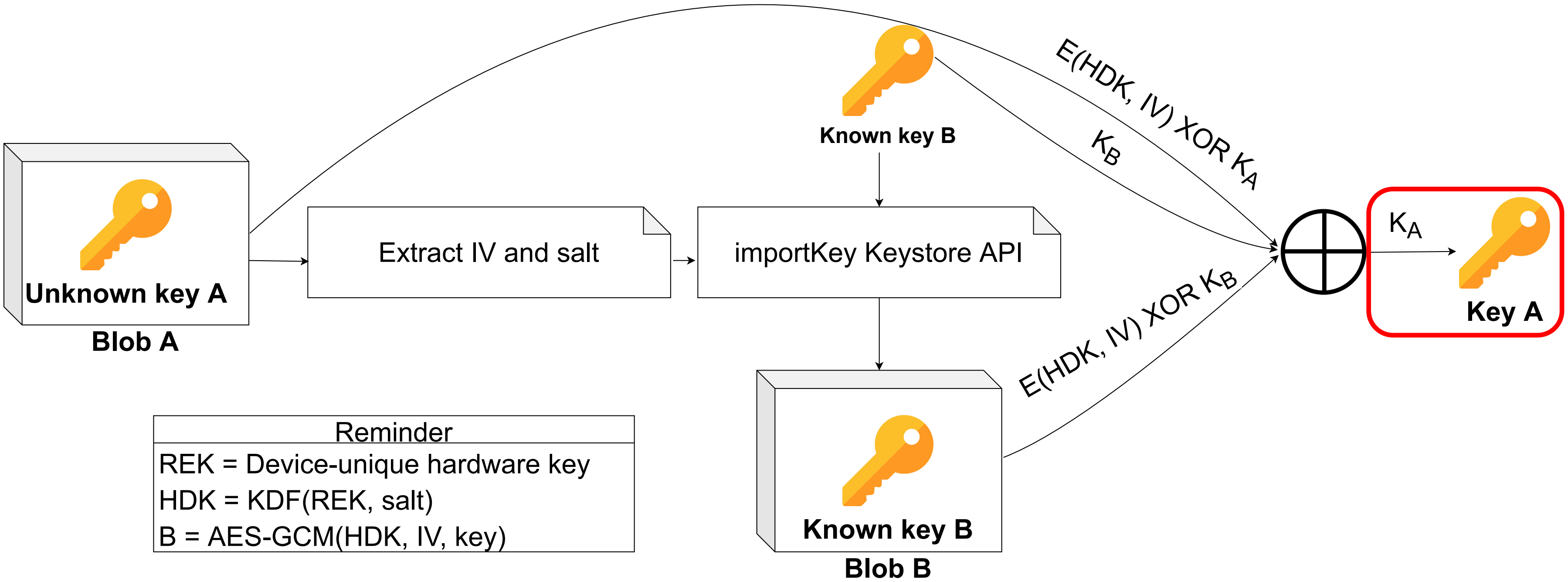
- The Android client can control the salt -> key reuse
- The Android client can control the IV -> IV reuse
- AES-GCM + key reuse + iv reuse -> decryption



Reminder
REK = Device-unique hardware key
HDK = KDF(REK, salt)
B = AES-GCM(HDK, IV, key)

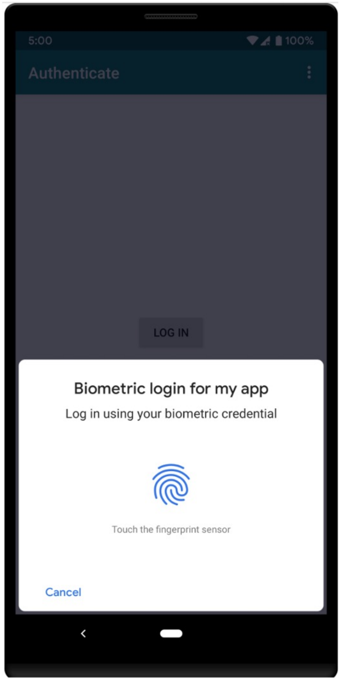
IV Reuse Attack (v15/v20-s9)

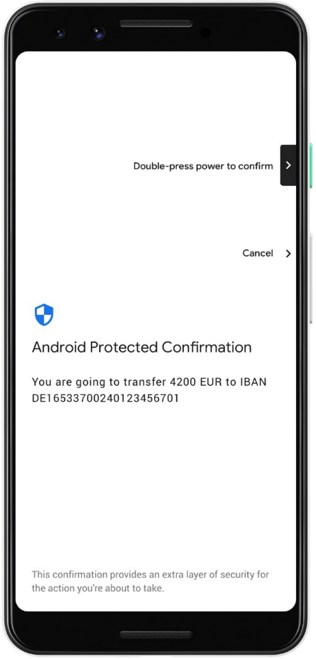
- The Android client can control the salt -> key reuse
- The Android client can control the IV -> IV reuse
- AES-GCM + key reuse + iv reuse -> decryption



Bypassing Authentication and Confirmation

The Keymaster TA can be used to enforce restrictions on the use of cryptographic keys to prevent misuse of the keys without the user's consent or knowledge





Images from Android Developers Blog

Downgrade Attack

- V20-s10 has randomized salt –> setting the IV is not enough

v20-s10 blob
"MDFPP HW Keymaster HEK v20\x00"
root_of_trust
"ID"
"\x02\x00\x00\x00"
"id"
"DATA"
"\x04\x00\x00\x00"
"data"
integrity_flags
hek_randomness

Downgrade Attack

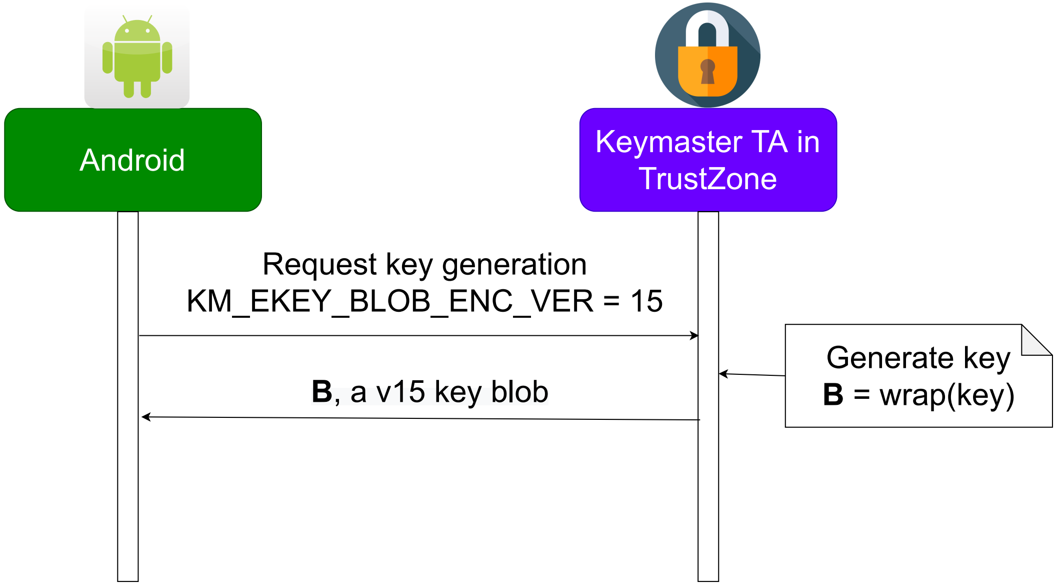
- V20-s10 has randomized salt –> setting the IV is not enough
- **Latent code** allows creation of v15 blobs

v20-s10 blob
"MDFPP HW Keymaster HEK v20\x00"
root_of_trust
"ID"
"\x02\x00\x00\x00"
"id"
"DATA"
"\x04\x00\x00\x00"
"data"
integrity_flags
hek_randomness

Downgrade Attack

- V20-s10 has randomized salt –> setting the IV is not enough
- **Latent code** allows creation of v15 blobs
- A privileged attacker can exploit this to force all new blobs to version v15

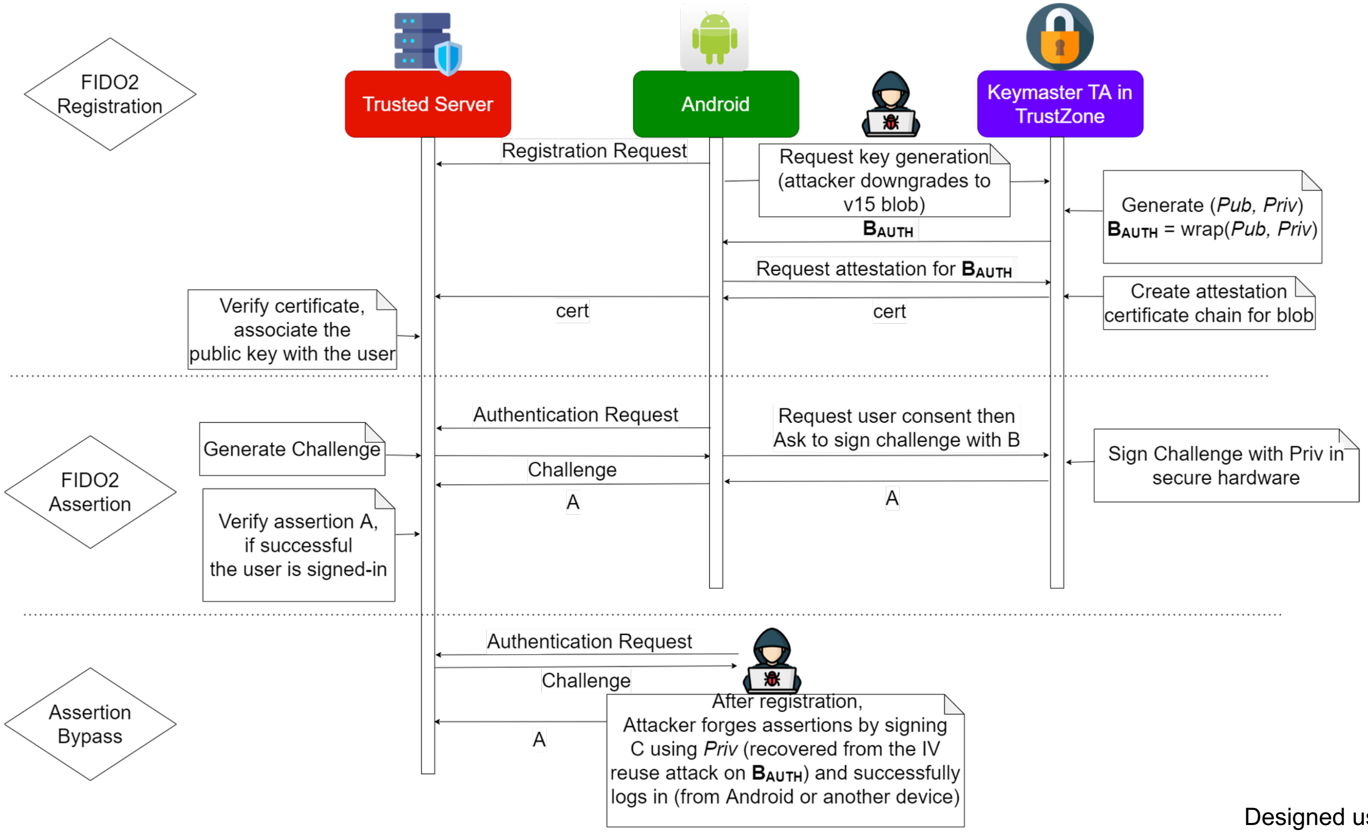
v20-s10 blob
"MDFPP HW Keymaster HEK v20\x00"
root_of_trust
"ID"
"\x02\x00\x00\x00"
"id"
"DATA"
"\x04\x00\x00\x00"
"data"
integrity_flags
hek_randomness



Agenda

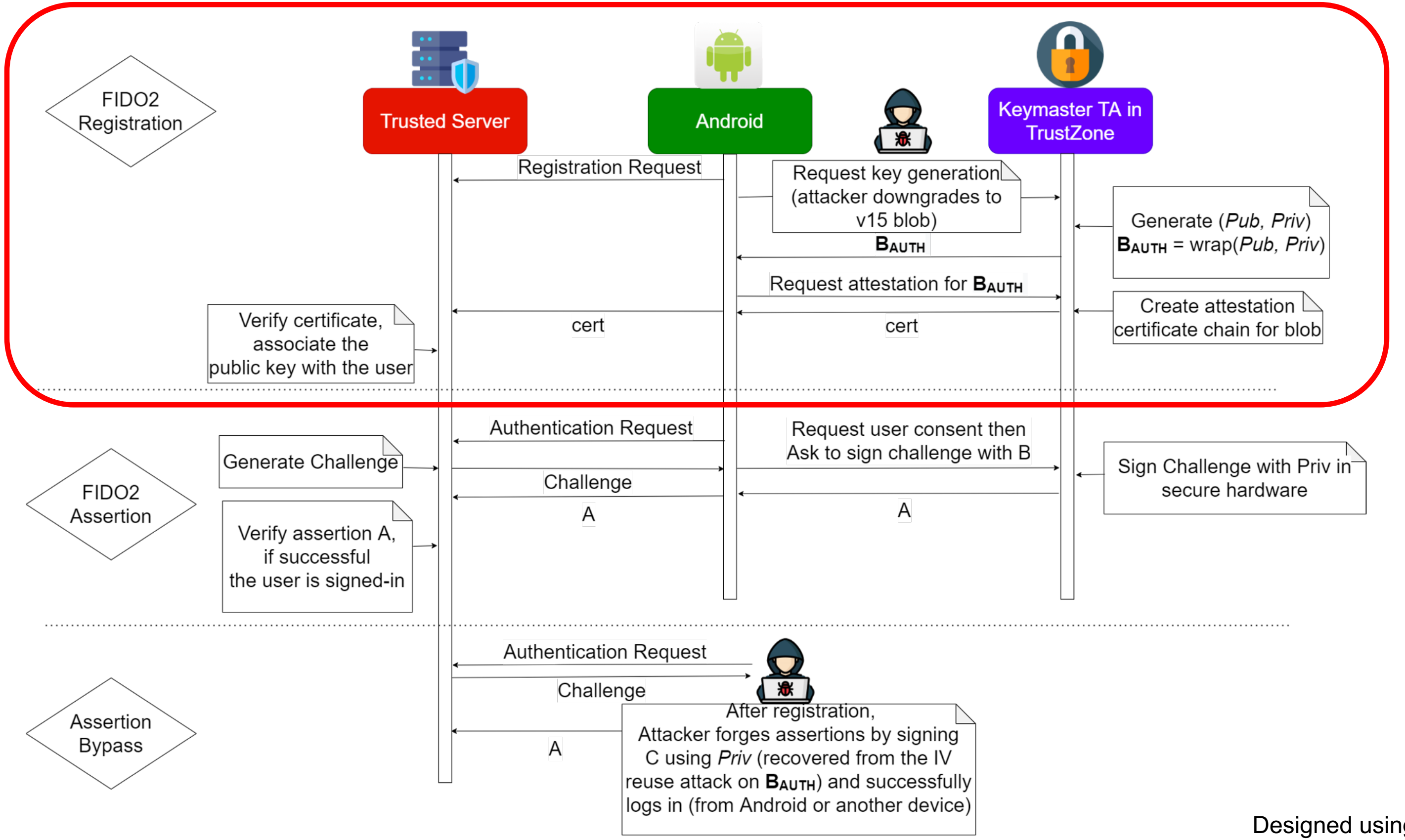
Introduction	Background and motivation
Keymaster TA Analysis	Recovering hardware-protected keys
Implications	Breaking higher-level protocols
Discussion	Main takeaways from our research

Bypassing FIDO2 WebAuthn



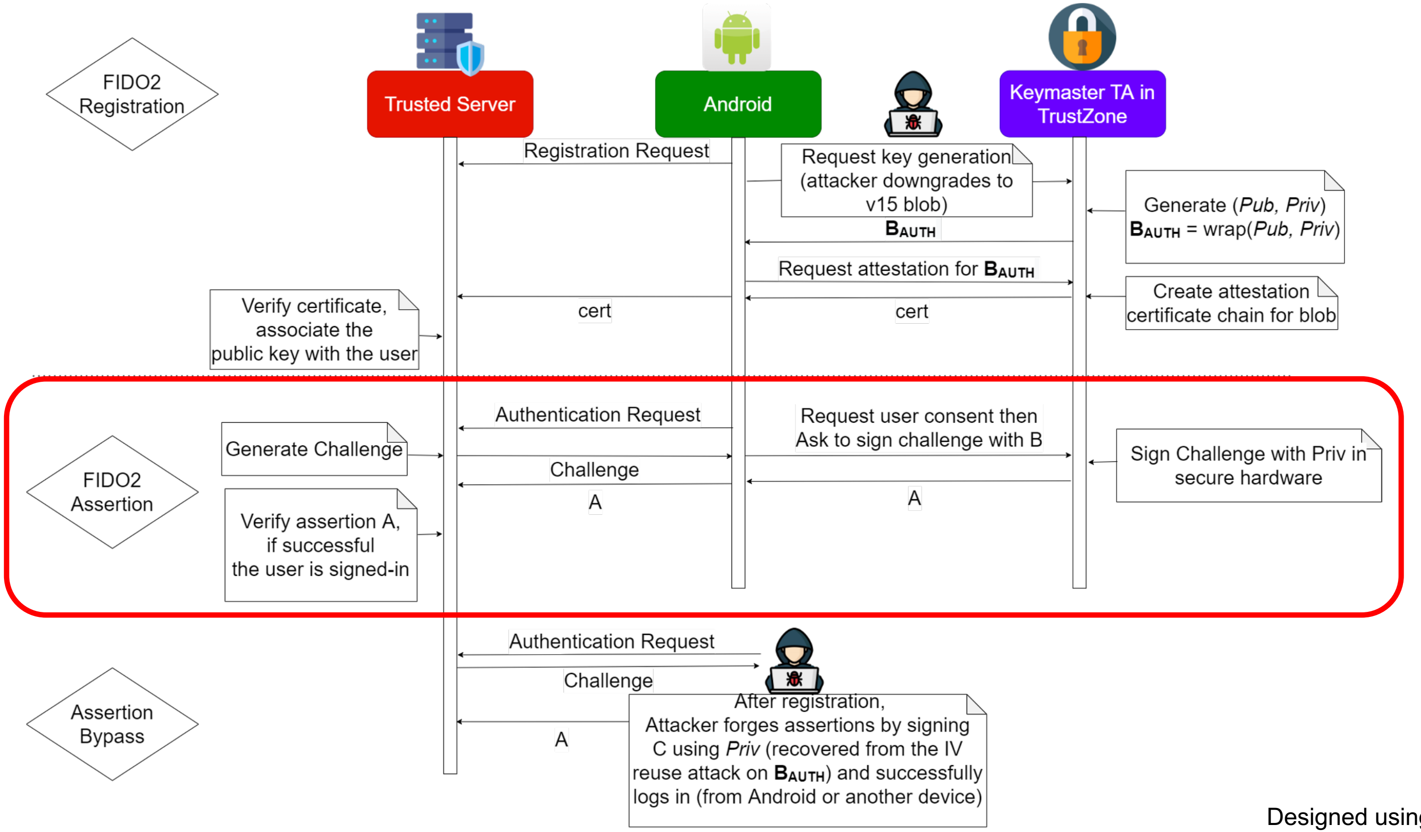
Designed using resources from Flaticon.com

Bypassing FIDO2 WebAuthn



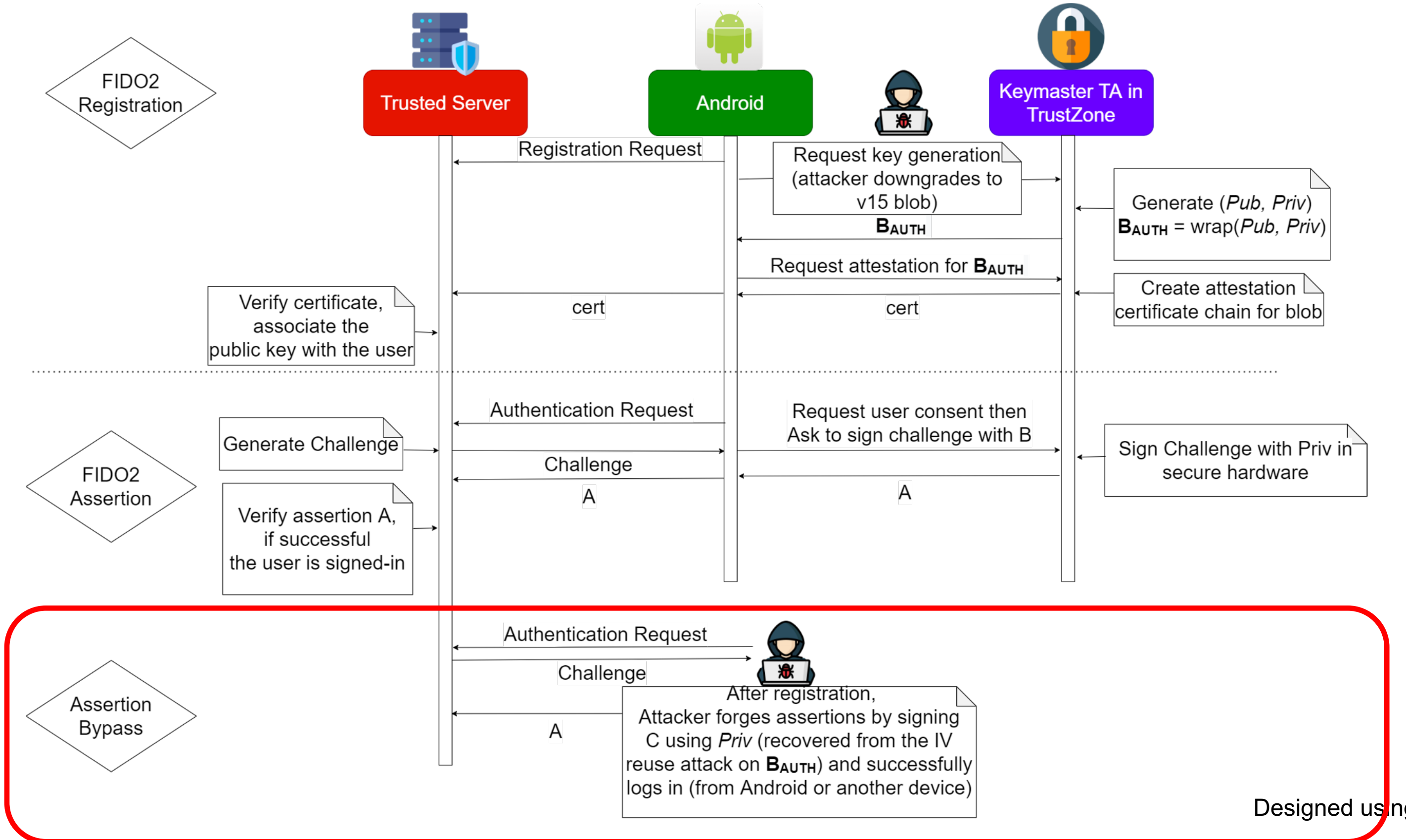
Designed using resources from Flaticon.com

Bypassing FIDO2 WebAuthn



Designed using resources from Flaticon.com

Bypassing FIDO2 WebAuthn



Designed using resources from Flaticon.com

Bypassing FIDO2 WebAuthn Demo #1

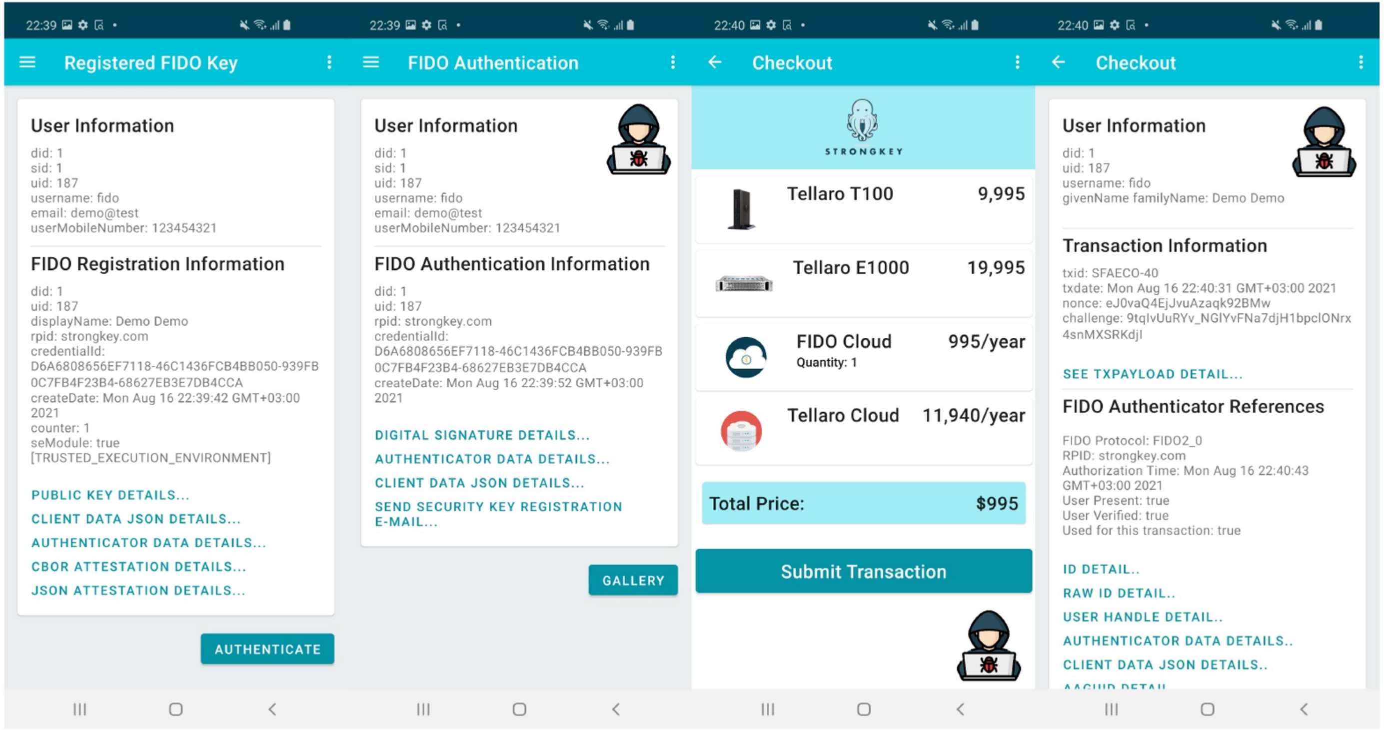
```
beyond1:/data/local/tmp # ./gdbserver --attach :1337 $(pidof android.hardware.keymaster@4.0-service)
Attached; pid = 5190
Listening on port 1337
```

(a) Attaching a GDB debugger to the Keymaster HAL process

```
Breakpoint 2, 0x00000077dae6e514 in nwd_generate_key () from target:/vendor/lib64/libkeymaster_helper_vendor.so
intercepted request to nwd_generate_key
copy old key parameters to new buffer
$1 = 0x7759c24000
$2 = 0x7759c24000
add new parameter (KM_EKEY_BLOB_ENC_VER, 15)
switch to new parameters - this forces the generation of a v15 blob
Breakpoint 4, 0x00000077dae6e544 in nwd_generate_key () from target:/vendor/lib64/libkeymaster_helper_vendor.so
dump the key blob that the keymaster returned
start 0x7759c3b280, end 0x7759c3b4d2, len 252
dumped to result.bin
```

(b) During registration, the GDB script performs the downgrade attack

Bypassing FIDO2 WebAuthn Demo #2



(c) Registration success (d) Authentication success (e) Checkout example (f) Re-authentication success

Responsible Disclosure

- We reported our IV reuse attack on S9 to Samsung in May 2021
  - According to Samsung, the list of patched devices includes: S9, J3 Top, J7 Top, J7 Duo, TabS4, Tab-A-S-Lite, A6 Plus, A9S

SVE-2021-21948 (CVE-2021-25444): IV reuse in Keymaster TA

Severity: High
Affected versions: O(8.1), P(9.0), Q(10.0)
Reported on: May 25, 2021
Disclosure status: Privately disclosed.
An IV reuse vulnerability in keymaster prior to SMR AUG-2021 Release 1 allows decryption of custom keyblob with privileged process.
The patch prevents reusing IV by blocking addition of custom IV.

Responsible Disclosure

- We reported our IV reuse attack on S9 to Samsung in May 2021
    - According to Samsung, the list of patched devices includes: S9, J3 Top, J7 Top, J7 Duo, TabS4, Tab-A-S-Lite, A6 Plus, A9S
- We reported the downgrade attack on S10, S20 and S21 in July 2021

SVE-2021-21948 (CVE-2021-25444): IV reuse in Keymaster TA

Severity: High
Affected versions: O(8.1), P(9.0), Q(10.0)
Reported on: May 25, 2021
Disclosure status: Privately disclosed.
An IV reuse vulnerability in keymaster prior to SMR AUG-2021 Release 1 allows decryption of custom keyblob with privileged process.
The patch prevents reusing IV by blocking addition of custom IV.

SVE-2021-22658 (CVE-2021-25490): Downgrade attack in Keymaster TA

Severity: High
Affected versions: P(9.0), Q(10.0), R(11.0)
Reported on: July 16, 2021
Disclosure status: Privately disclosed.
A keyblob downgrade attack in keymaster prior to SMR Oct-2021 Release 1 allows attacker to trigger IV reuse vulnerability with privileged process.
The patch removes the legacy implementation for minor keyblob.

Samsung Patch #1

In August 2021 Samsung assigned CVE-2021-25444 with High severity and released a patch that removes the option to add a custom IV from the API.

```c
static struct blob *_swd_get_iv(KM_PARAM_SET *par)
{
    static uint8_t buf[KM_IV_LEN_DEFAULT];
    static struct blob b;
    int ret = 0, is_enc_pw_key = 0;

    b.data = buf;
    b.len = sizeof(buf);
    if (par && (ret = km_get_tag(par, KM_TAG_EKEY_BLOB_IV, 0, &b)) < 0)
        LOGW("km_get_tag() failed");
    if (par && (km_get_km_param_str(par, KM_TAG_EKEY_BLOB_PASSWORD, 0) != NULL))
        is_enc_pw_key = 1;

    LOGD("%s IV", (ret == 1 && is_enc_pw_key == 1) ? "custom" : "generated");
    if (ret == 1 && is_enc_pw_key == 1)
    {
        // [SI-21948] Keymaster TA IV Reuse vulnerability
        // Not allow to add custom iv, only encrypt_password_hwkey is allowed.
        return &b;
    }

    if (!RAND_bytes(b.data, b.len))
    {
        LOGE("swd_get_iv() failed");
        return NULL;
    }

    return &b;
}
```

```c
int swd_get_iv(KM_PARAM_SET *par, ASN1_OCTET_STRING *iv)
{
/*
    // [SI-21948] Keymaster TA IV Reuse vulnerability
    // Not allow to add custom iv
    int ret = 0;

    if (par && (ret = km_get_tag_str(par, KM_TAG_EKEY_BLOB_IV, 0, iv)) < 0)
        LOGW("km_get_tag() failed");

    if (ret == 1)
        return 0;
*/
    (void)par;
    if (!RAND_bytes(iv->data, iv->length)) {
        LOGE("RAND_bytes() failed");
        return -1;
    }

    return 0;
}
```

Samsung Patch #2

In October 2021 Samsung assigned CVE-2021- 25490 with High severity and released a patch that completely removes the legacy key blob implementation.

```
static int _swd_get_salt(struct blob *rot, uint32_t integrity_status, struct blob *client_id,
                         struct blob *app_data, struct blob *ukdm, uint8_t blob_ver,
                         ASN1_OCTET_STRING *salt)
{
    uint32_t clid_size, appd_size;
    const struct blob **set;
    size_t len;
    int i;
    uint8_t cbd_buf[CRYPTO_BOUNDING_BUFF_LEN] = { 0 };
    struct blob cbd = { cbd_buf, 0 };
    struct blob app_data_size = { NULL, 0 };
    struct blob client_id_size = { NULL, 0 };
    struct blob integrity = { NULL, 0 };
    struct blob _rot = *rot;

    const struct blob *salt_seq_2_0[] = {
        &context_salt_2_0, &_rot,      NULL, /* ID label */
        &client_id_size,   client_id, NULL, /* DATA label */
        &app_data_size,    app_data,  &integrity, ukdm,
    };
    size_t salt_seq_len_2_0 = sizeof(salt_seq_2_0) / sizeof(salt_seq_2_0[0]);

    if (blob_ver == KM_KEY_BLOB_ENC_VER) {
        set = salt_seq_2_0;
        len = salt_seq_len_2_0;
    } else {
        LOGE("invalid blob_ver: %d", blob_ver);
        return -1;
    }
```

Agenda

Introduction	Background and motivation
Keymaster TA Analysis	Recovering hardware-protected keys
Implications	Breaking higher-level protocols
Discussion	Main takeaways from our research

Low-Level Cryptographic Issues

Allowing client to set IV

Allowing client to set encryption version

Latent code in security-critical application

Encryption version persists across "upgrades"

Low-Level Cryptographic Issues

✗ Allowing client to set IV	✗ Allowing client to set encryption version	✗ Latent code in security-critical application	✗ Encryption version persists across "upgrades"
✓ Using random IV	✓ Disallowing choice encryption version	✓ Reducing attack surface in security-critical application	✓ Always using the latest encryption version

The Gap in Composability

Key attestation does not commit to the cryptographic method

Closed vendor-specific implementation

The Gap in Composability

Key attestation does not commit to the cryptographic method

Closed vendor-specific implementation

Including encryption method in attestation certificate

Uniform open-standard by Google for the Keymaster HAL and TA

Conclusions

- Vendors including Samsung and Qualcomm maintain secrecy around their implementation and design of TZOSs and TAs.
- Through our analysis we unveiled severe cryptographic design flaws
- We show how to exploit the design flaws to break higher level protocols

Proprietary OEM implementation + Pitfalls of cryptographic APIs = Severe design flaws

Conclusions

- Vendors including Samsung and Qualcomm maintain secrecy around their implementation and design of TZOSs and TAs.
- Through our analysis we unveiled severe cryptographic design flaws
- We show how to exploit the design flaws to break higher level protocols
- All of those issues could have been avoided with an open-standard design

Proprietary OEM implementation + Pitfalls of cryptographic APIs = Severe design flaws

Conclusions

- Vendors including Samsung and Qualcomm maintain secrecy around their implementation and design of TZOSs and TAs.
- Through our analysis we unveiled severe cryptographic design flaws
- We show how to exploit the design flaws to break higher level protocols
- All of those issues could have been avoided with an open-standard design
- Using the fragile AES-GCM deserves discussion - after decades of IV reuses in real-world systems

Proprietary OEM implementation + Pitfalls of cryptographic APIs = Severe design flaws

Any questions?

Extended paper: https://eprint.iacr.org/2022/208.pdf

Designed using resources from Flaticon.com