armasuisse
Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

CYD
CYBER DEFENCE CAMPUS

u^b UNIVERSITÄT BERN

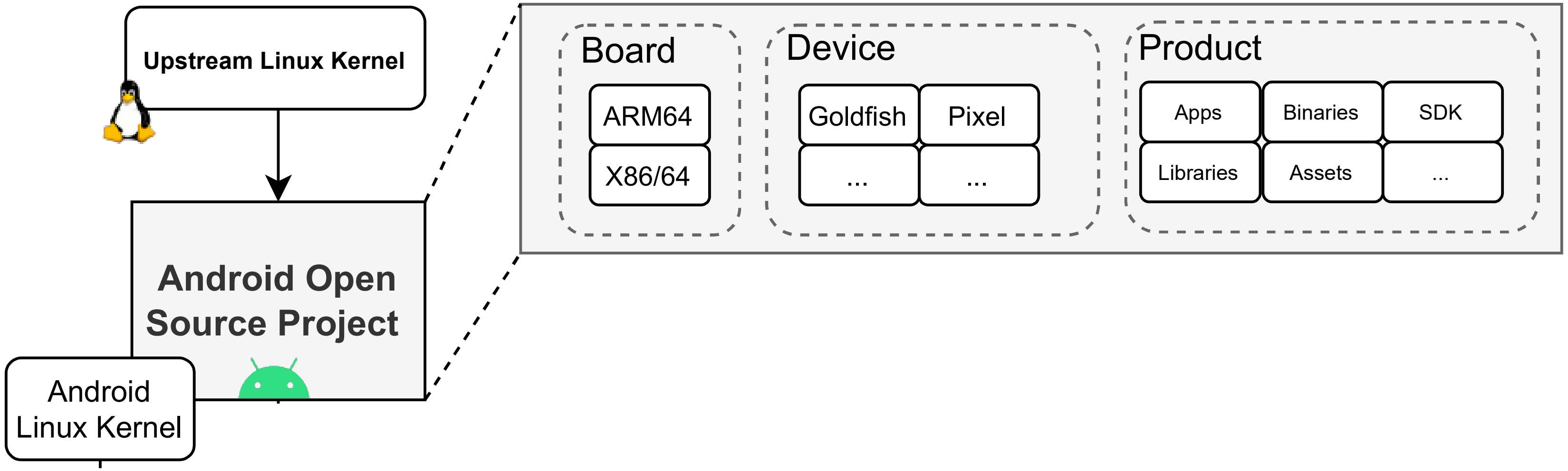
zh aw

Runtime Vulnerability Detection in Android pre-installed Apps

Thomas Sutter

June 2025

Fragementation



Customization Options:

- Boards
- Devices
- Products
- Regions
- Language
- Carriers
- …

Vendors customize Android extensively.
While this enables product differentiation, it results in fragmentation of the OS.

Android devices are everywhere…

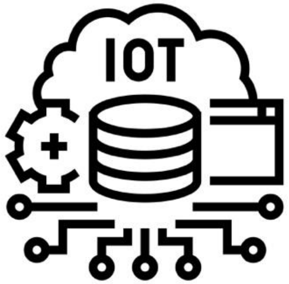
TV-Boxes

Smartphones & Tablets

Car Infotainment System

Point-of-Sales Systems

Wearables



- Medical Devices
- Smart Home Systems
- Vending Machines
- Kiosks
- Industrial Control Panels

…

Pre-Installed Apps

Framework Core Apps
- Settings
- Launcher
- Phone
- Contacts
- PermissionController
- Cellbroadcast
- Bluetooth
- Wifi
- NFC
- PackageInstaller
- SystemUI
- Telecom
- ContactsProvider
- MediaProvider
- LatinIME
- Dialer
- NetworkStack
- Tethering

…

Vendor Apps
- Browser
- Debug Info
- Camera
- Audio Recorder

…

Manufacturer Apps
- Google Home
- Google Playstore
- Google Play Protect
- AI Assistant

…

Third-Party Apps
- Office
- Social Media
- Weather
- News

…

Facts:

- **Android devices have between 50 to 1’000 apps pre-installed.**
- **Most apps are not public available on any app store**
- **Pre-Installed apps can’t be removed by users and they are not frequently updated**
- **Dynamic Analysis is challenging**

Challenges in testing pre-installed Apps

Platform Constraints

□ System Partition Integration

□ Singleton Apps

□ Framework Dependencies

Security Constraints

□ Android Verified Boot

□ Read-Only File Systems

□ SELinux Policies

□ Intergrity Protections (Signatures)

□ Permission Whitelisting

□ Device Attestation

□ No Root Rights on Device

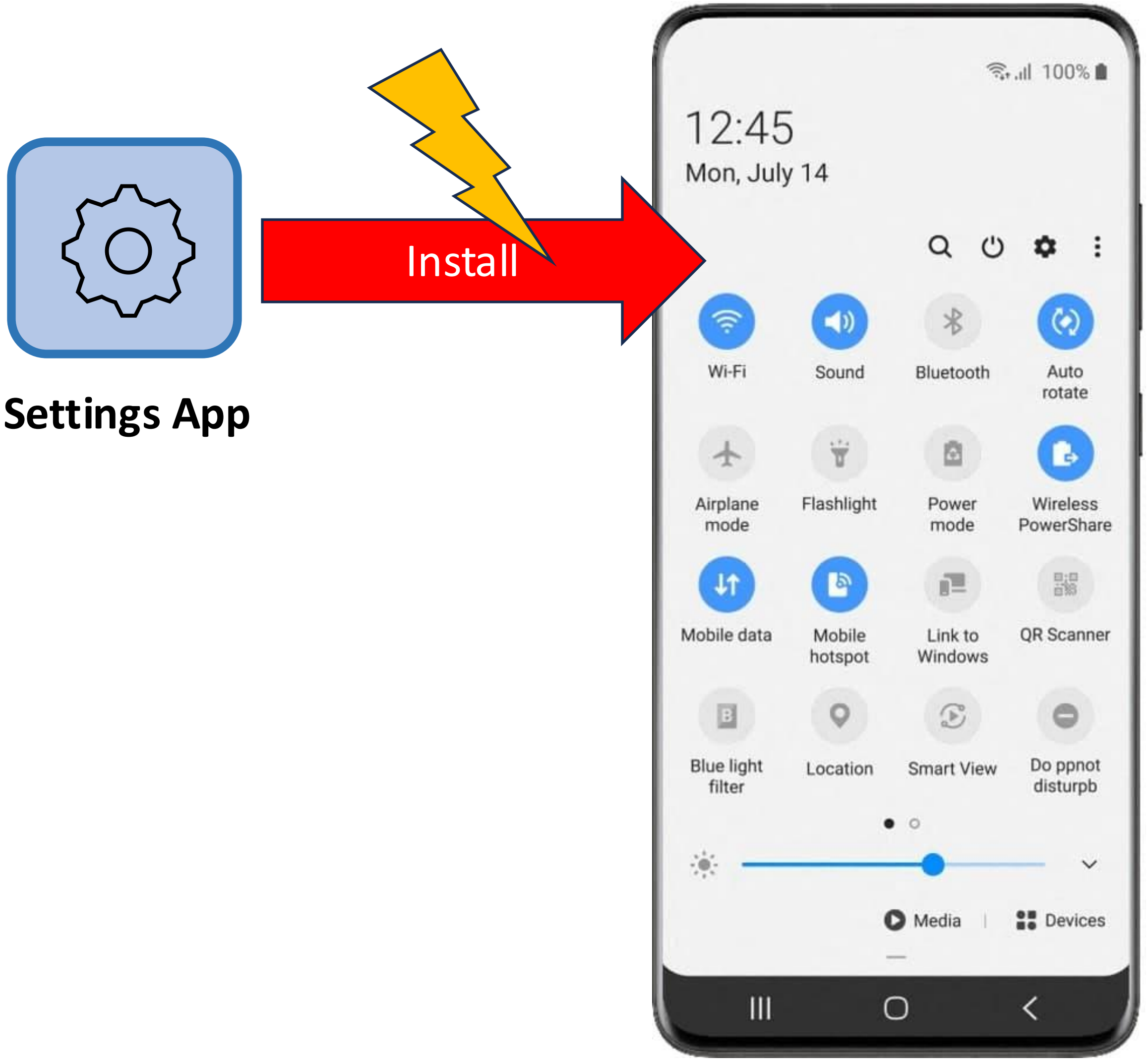
App Constraints

□ Inter-App Dependencies (Collusion)

□ File and Native-Library Dependencies

□ External Service Dependencies

□ Environment Checks & Anti-Analysis



Consequences

Physical device usage

- Often works for security testing with manual effort
- Does not scale well (costs and availability)
- Limited to devices with root access
  - Rooting is often not trivial and laboursome
- Security Tests might brick the device

Fallback to static analysis methods

- Often misses runtime vulnerabilities
- Challenges with dynamic code updates
- Code may be obfuscated or packed

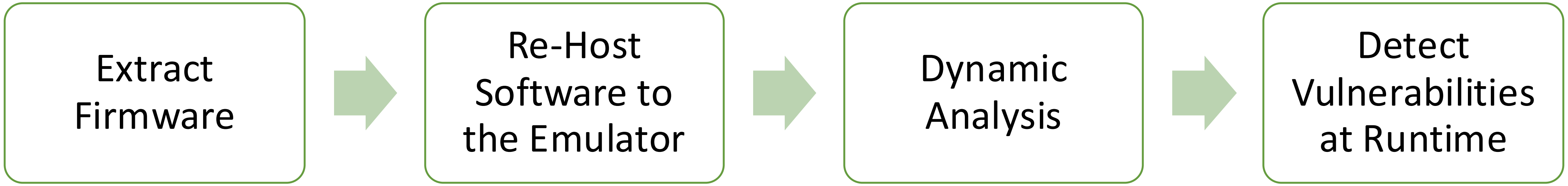
Objective

Developing a method to target real-world vendor-modified pre-installed apps **without** using physical devices.

Purpose → Testing vendor-customized Android pre-installed apps to find runtime vulnerabilities

Idea → Re-Host the pre-installed applications using **emulation**

The Re-Hosting Idea



Extract Firmware

Re-Host Software to the Emulator

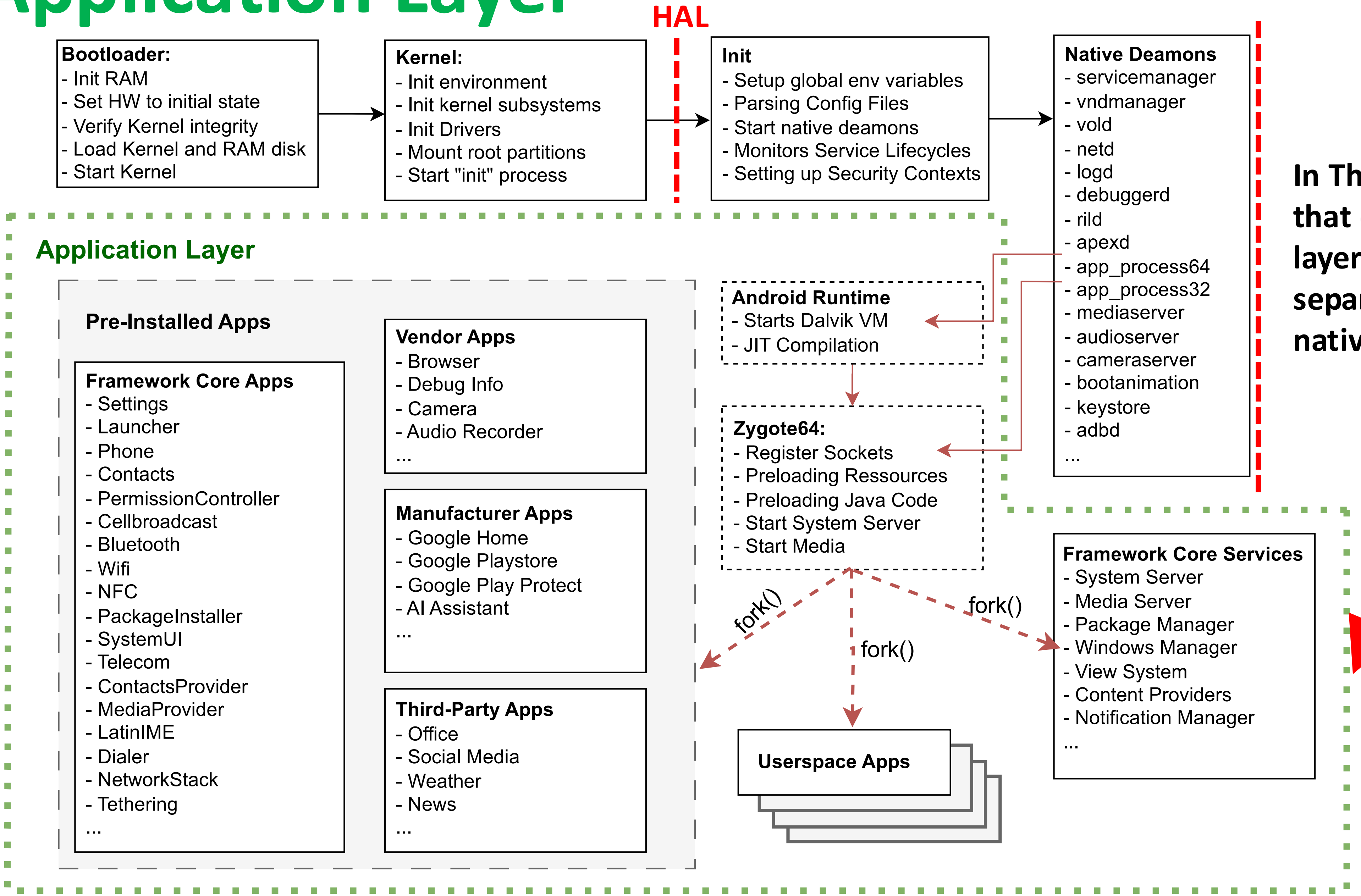
Dynamic Analysis

Detect Vulnerabilities at Runtime

Are current emulators capable of executing ARM code?

Which software components do we need to re-host to correctly execute the pre-installed apps?

Application Layer



HAL

Bootloader:
- Init RAM
- Set HW to initial state
- Verify Kernel integrity
- Load Kernel and RAM disk
- Start Kernel

Kernel:
- Init environment
- Init kernel subsystems
- Init Drivers
- Mount root partitions
- Start "init" process

Init
- Setup global env variables
- Parsing Config Files
- Start native deamons
- Monitors Service Lifecycles
- Setting up Security Contexts

Native Deamons
- servicemanager
- vndmanager
- vold
- netd
- logd
- debuggerd
- rild
- apexd
- app_process64
- app_process32
- mediaserver
- audioserver
- cameraserver
- bootanimation
- keystore
- adbd

...

Application Layer

Pre-Installed Apps

Framework Core Apps
- Settings
- Launcher
- Phone
- Contacts
- PermissionController
- Cellbroadcast
- Bluetooth
- Wifi
- NFC
- PackageInstaller
- SystemUI
- Telecom
- ContactsProvider
- MediaProvider
- LatinIME
- Dialer
- NetworkStack
- Tethering

...

Vendor Apps
- Browser
- Debug Info
- Camera
- Audio Recorder

...

Manufacturer Apps
- Google Home
- Google Playstore
- Google Play Protect
- AI Assistant

...

Third-Party Apps
- Office
- Social Media
- Weather
- News

...

Android Runtime
- Starts Dalvik VM
- JIT Compilation

Zygote64:
- Register Sockets
- Preloading Ressources
- Preloading Java Code
- Start System Server
- Start Media

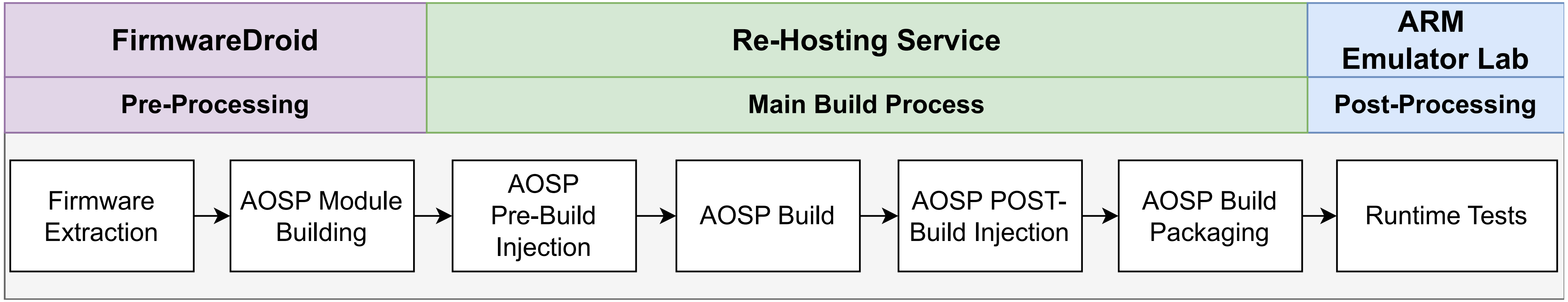
Userspace Apps

Framework Core Services
- System Server
- Media Server
- Package Manager
- Windows Manager
- View System
- Content Providers
- Notification Manager

...

In Theory, but practice shows that due to fragmentation the layer is not that clearly separated from the native deamons or kernel

How to Re-Host the Application Layer?

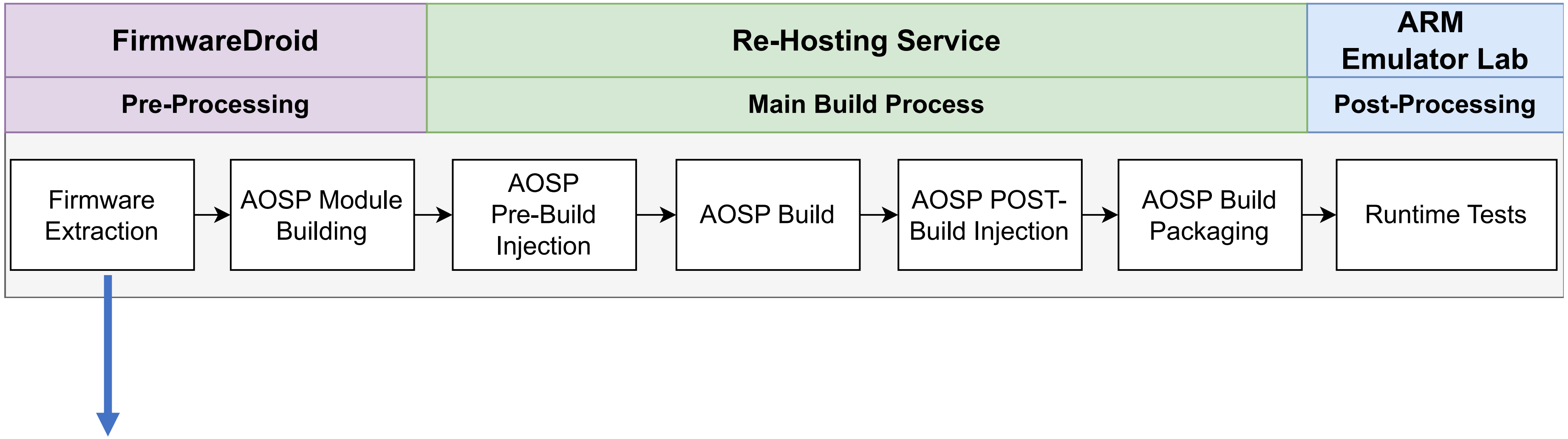


FirmwareDroid	Re-Hosting Service	ARM Emulator Lab
Pre-Processing	Main Build Process	Post-Processing

Firmware Extraction → AOSP Module Building → AOSP Pre-Build Injection → AOSP Build → AOSP POST-Build Injection → AOSP Build Packaging → Runtime Tests

Our approach is based on the default build process of the Android Open Source Project

Re-Hosting Prototype



FirmwareDroid

Pre-Processing

Re-Hosting Service

Main Build Process

ARM Emulator Lab

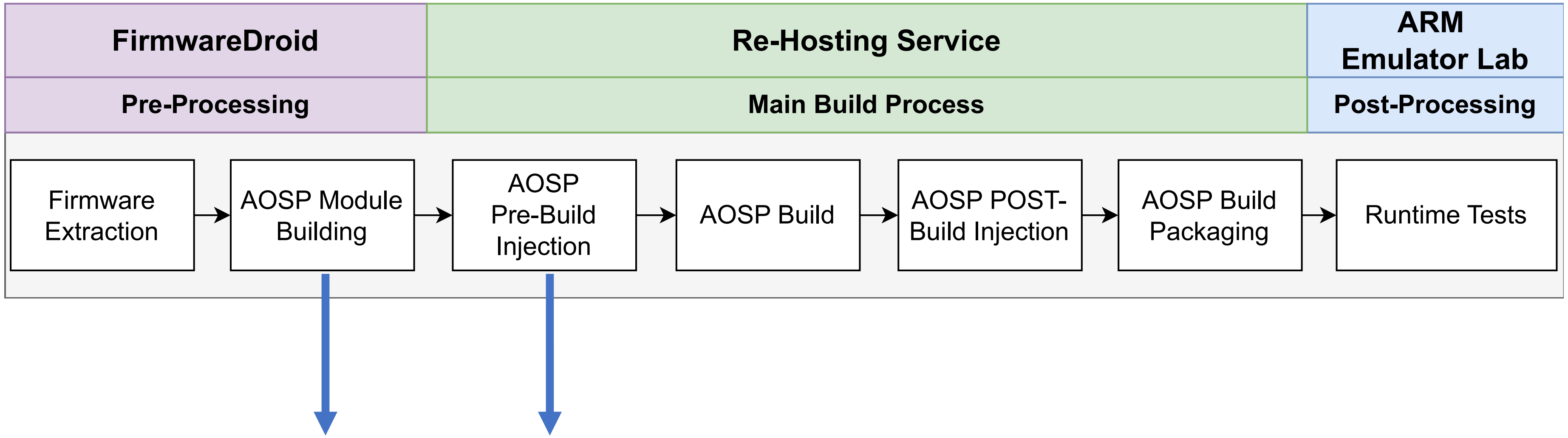
Post-Processing

Firmware Extraction → AOSP Module Building → AOSP Pre-Build Injection → AOSP Build → AOSP POST-Build Injection → AOSP Build Packaging → Runtime Tests

Extended FirmwareDroid to be able to extract firmware from major manufactures (Google, Samsung, Xiaomi, Vivo, Huawei, …)

- Samsung and Huawei firmware are still often troublesome due to custom image formats

Re-Hosting Prototype



FirmwareDroid | Re-Hosting Service | ARM Emulator Lab

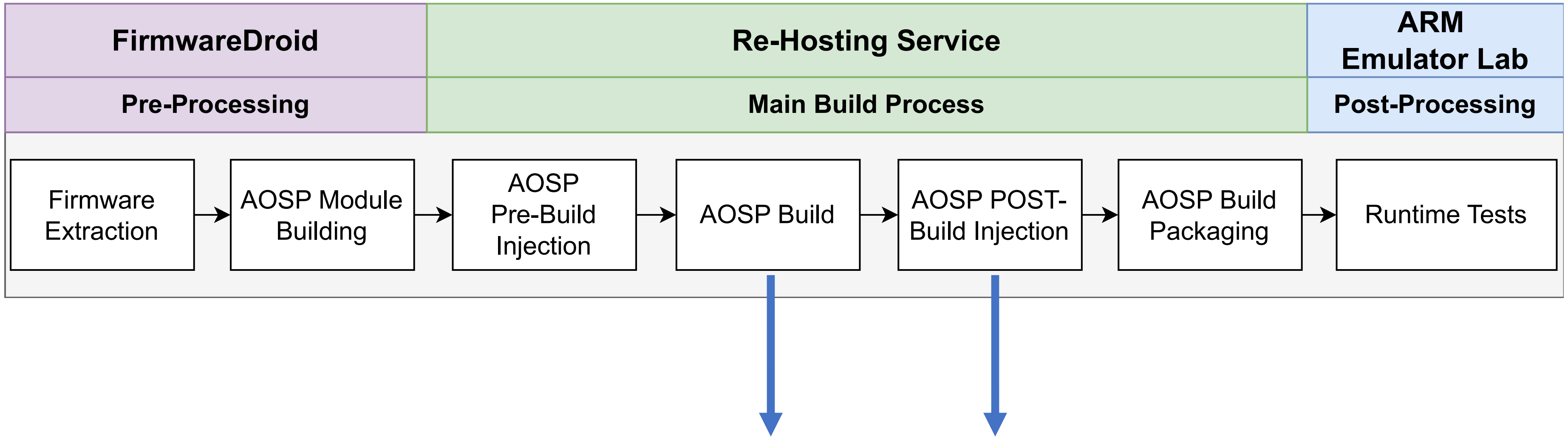
Pre-Processing | Main Build Process | Post-Processing

Firmware Extraction → AOSP Module Building → AOSP Pre-Build Injection → AOSP Build → AOSP POST-Build Injection → AOSP Build Packaging → Runtime Tests

Develop a module builder that packs pre-installed apps (.apk) and native libraries (.so) into build modules (.mk, .bp)

- **Allows us to inject pre-built files into the normal build process of AOSP**
  - Automates the signing process for apps
    - Correct certificate selection (platform, media, networkstack, …) is done as well by FMD
- **AOSP does not allow us to overwrite the AOSP framework with our pre-built files (e.g., .jar, elf, …)**

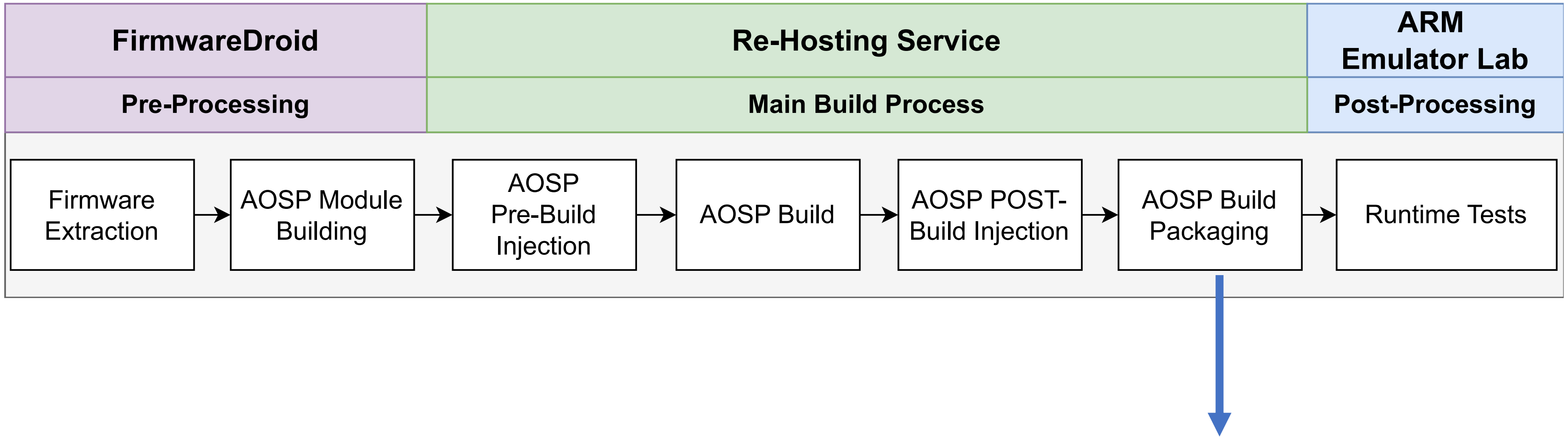
Re-Hosting Prototype



Injecting all AOSP build incombatible files after the main build:

- Developed an Algorithm to replace intermediate files
    - **Allows us to overwrite framework components (framework.jar, services.jar, APEX, …)**
        - **Including all pre-installed apps**
- Rule-based approach to inject or replace files at will
    - **Builds might break at runtime depending on the injected files**
        - **Modifications to certain core binaries or libraries are unlikely to fully work**

Re-Hosting Prototype



FirmwareDroid

Pre-Processing

Re-Hosting Service

Main Build Process

ARM Emulator Lab

Post-Processing

Firmware Extraction → AOSP Module Building → AOSP Pre-Build Injection → AOSP Build → AOSP POST-Build Injection → AOSP Build Packaging → Runtime Tests

Packing using the default command of AOSP to build emulator images:

- Converts the image to an ARM emulator image
- **Signs images correctly -> Passes Android Verified Boot (AVB)**
- **Packes all the files into an image that is compatible with the emulator**

Our Contribution

Work in Progress

Platform Constraints

- ✓ System Partition Integration
- ✓ Singleton Apps
- ✓ Framework dependent

Security Contstraints

- ✓ Android Verified Boot
- ✓ Read-Only File Systems
- ✓ SELinux Policies
- ✓ Intergrity protections (Signatures)
- ✓ Permission Whitelisting
- □ Device Attestation
- ✓ No Root Rights on Device

App Constraints

- ✓ Inter-App Dependencies (collusion)
- ✓ File and Native-Library Dependencies
- □ External Service Dependencies
- □ Environment Checks & Anti-Analysis

Limitations

Trade-offs and Limitations

- No Kernel Re-Hosting
- Limited Re-hosting capabilities
  - **Selective** injection of files
    - No full re-hosting of the userspace
  - Builds can easily break at startup
- Device Attestation is still a problem

- Slow Build Times (around 90mins per build)
  - Slows down development progress significantly

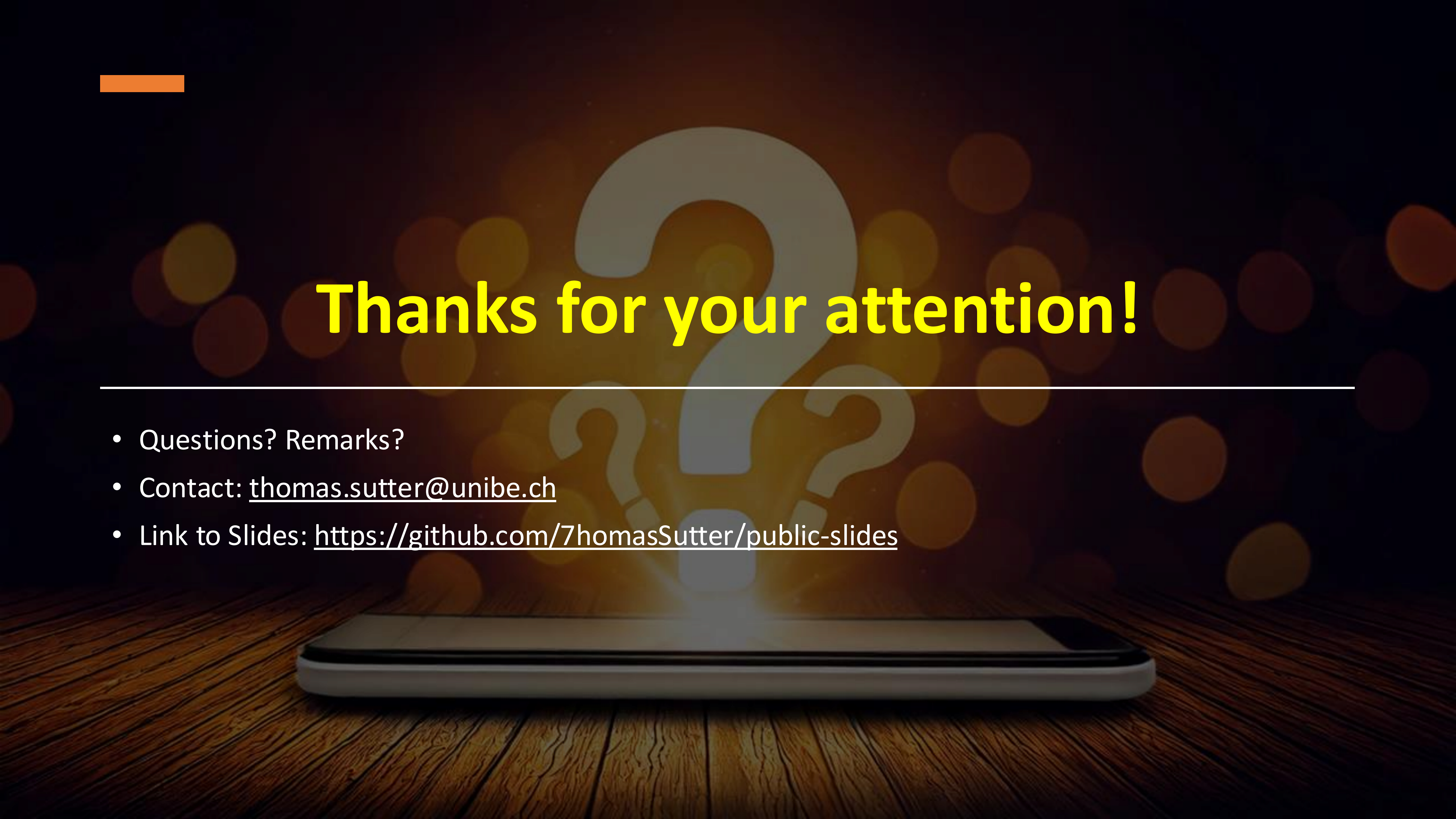
Takeaways and What's Next

Takeaways:

- ➢ Emulation of real-world code (pre-installed apps) is feasible
- ➢ Paralellization and scaling on the ARM Emulator works
- ➢ Re-Hosting is heavily device specific due to fragmentation of the Android OS

Next:

- ❑ Evaluation of our approach on around 200 firmware samples
- ❑ Instrumentation Integration
  - ❑ Integrate Network Monitoring
    - ❑ Open Apps Ports (TCP / UDP)
  - ❑ Integrate API Call Tracing
    - ❑ Identify the usage of hidden framework APIs
    - ❑ Analyse custom framework permissions
    - ❑ Analyse exposed pre-installed apps services



Thanks for your attention!

- Questions? Remarks?
- Contact: thomas.sutter@unibe.ch
- Link to Slides: https://github.com/7homasSutter/public-slides