

# Mobile Phone Security: Android (and others)

19. October 2011, 18:30

NIG Universität Wien

**CERT.at Security-Stammtisch**

**Prof. (FH) Priv.-Doz. DI Dr. René Mayrhofer**

Fachhochschule Hagenberg (Mobile Computing)

eSYS GmbH (Security Consultant)

[rene.mayrhofer@fh-hagenberg.at](mailto:rene.mayrhofer@fh-hagenberg.at)

[rene@mayrhofer.eu.org](mailto:rene@mayrhofer.eu.org)

The most profound technologies are those that **disappear**. They weave themselves **into the fabric of everyday life** until they are **indistinguishable** from it.

Mark Weiser, 1991, „The Computer for the 21st Century“

# The changed environment



# Problem areas

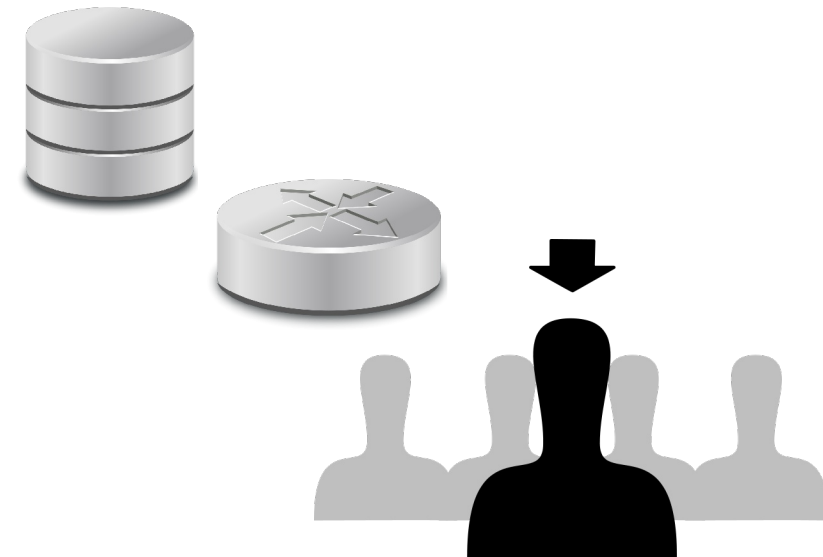
## Security for whom?

- Mobile device
- infrastructure
- User



## Security for what?

- (on-device) data
- (wireless) communication
- (user) privacy





# Data security on mobile devices

## Threats

- Mobile devices are complete computers, but with more (open) interfaces
- Malware
- Physical access
- Loss / theft and revocation
- Wireless networks
- (currently) barely any security measures



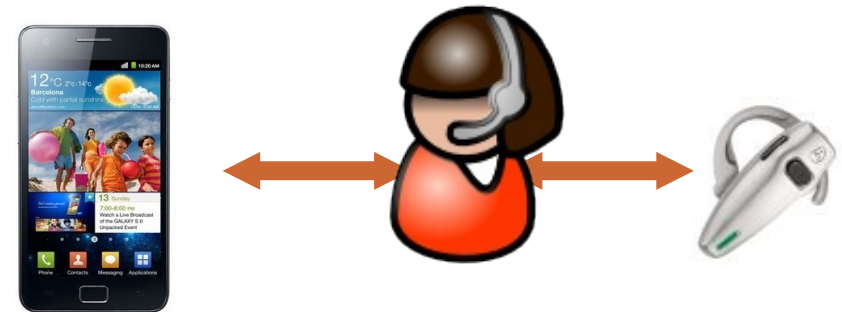
⇒ **Data security on devices is difficult and can only be guaranteed with major effort and/or severe restrictions for end-users.**



# Wireless communication

## Threats

- Eavesdropping
- Replay
- Modification, fabrication
- Faking, spoofing
- Denial of service
- Man-in-the-Middle
- Attacks on implementation bugs in devices
- Working around network blocks by using varying (wireless) network connectivity and anonymization techniques



⇒ **Wireless communication must always be assumed to be completely untrusted.**



# Mobile user privacy

## Threats

- Lots of personal data items on mobile device
  - Contacts
  - Calendar
  - Messages (email, SMS, instant messaging)
  - Captures images, videos
  - History (applications, web browser, search terms)
- Detailed profiling of end-users feasible and economically interesting
- Social engineering based on this data is made significantly easier
- Increasing number of security critical application scenarios
  - Credit card and electronic banking data
  - Health data
  - Location and communication patterns



# Proliferation of mobile platforms

- Symbian OS
- Windows Mobile / CE / Windows Phone 7
- Linux
  - LiMo, EZX, etc.
  - Android
  - Maemo / MeeGo / Tizen
  - bada
- Apple iPhone
- Blackberry
- Java 2 Micro Edition (J2ME)



# iPhone vs. Android security concepts

- **Closed** eco-system
- All applications need to go through Apple Store, checked by Apple
- **unless device has been "jailbroken"...**
- Sandboxing restricted to few applications (e.g. MobileSafari)
- Some exploit protection measures
  - Data Execution Prevention (DEP) using ARM XN feature
  - but no ASLR in kernel
- Some security APIs (key storage, RNG)
- Transparent device encryption **with key stored on device**
- **No further security measures**



- **Open** eco-system
- Arbitrary applications can be installed by the user (may need to enable "USB debugging for installation from PC or download on-device and install)
- **All applications are sandboxed**
- DRM for applications introduced recently (Android 2.2 and new Market API)
- No further security measures
  - no ASLR, no DEP
  - no on-device encryption



# Comparing exploit prevention techniques

Table: Exploit mitigation techniques

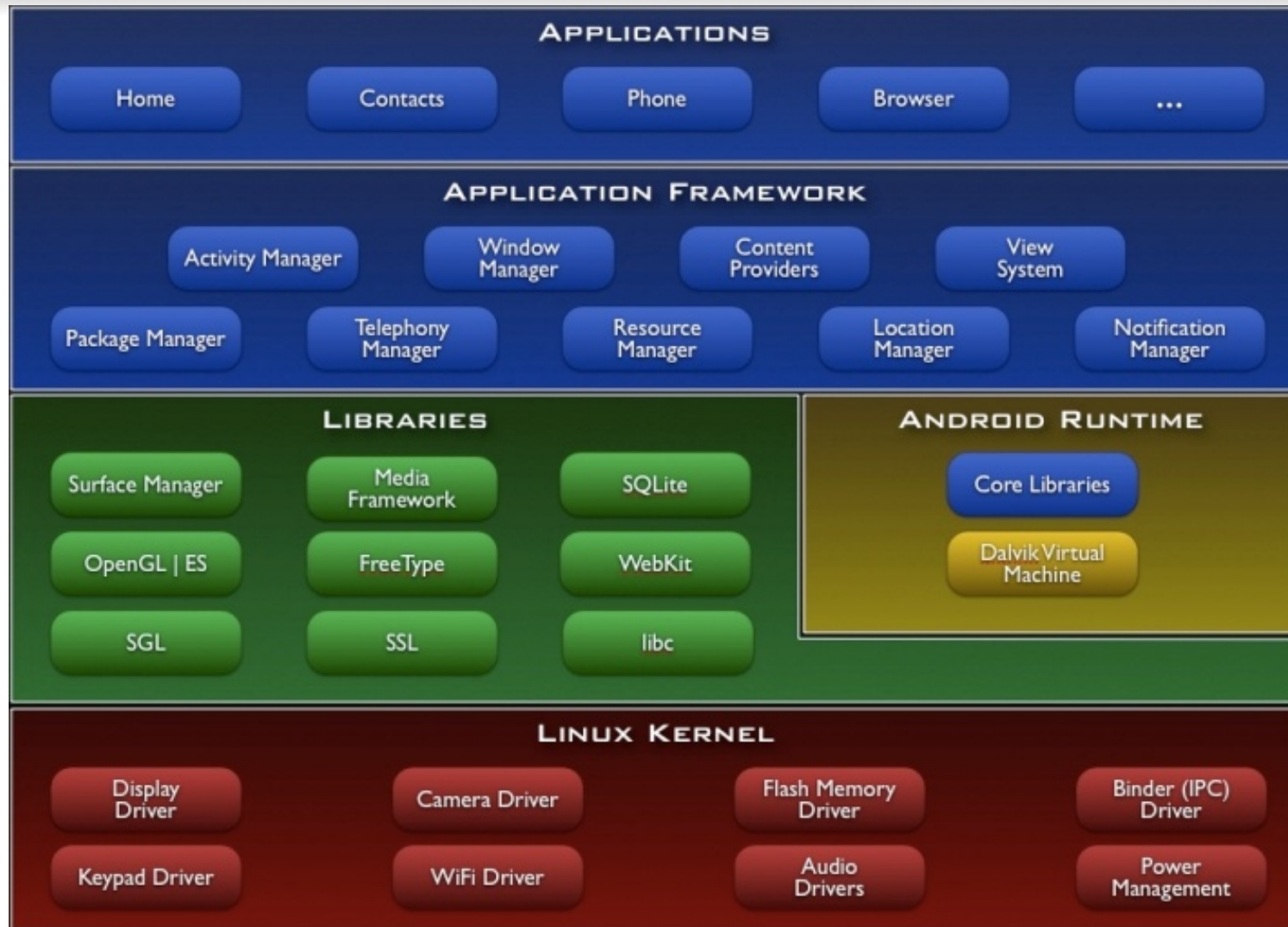
Protection	Android	W. Mobile	Iphone
Stack NX	-	-	Yes
Heap NX	-	-	Yes
Cookie	-	Yes, 16 bit	-
Random Libs	-	-	-
Random Stack	Yes	-	-
SEH	-	stack	-

Table by Nicolas Economou and Alfredo Ortega, Presentation "Smartphone (in)security" at CanSecWest 2009

# Comparing mobile platforms: summary

	<b>Android</b>	<b>iOS</b>	<b>Blackberry</b>	<b>Symbian</b>
Restricted to App Store	no	yes	no	no
Sandbox for applications	yes	some (Safari)	no (unknown)	no
Signed applications	yes	yes	yes	yes
Capabilities for applications	yes (all-or-nothing)	no	yes (configurable)	yes (configurable)
Garbage collection	yes (Java)	no (Objective C)	yes (Java)	no (C++)
<b>Exploit prevention</b>	no NX no ASLR	NX stack+heap no ASLR	unknown	no NX no ASLR
On-device encryption	no	yes, but problematic	yes	no

# Android Architecture



# Android Security Architecture

## Applications must be signed for installation

- may be self-signed by the developer, therefore no requirement for centralized application Q/A or control
- signature allows non-repudiability (if the public key/certificate is known)
- signature by same private key allows applications to share data and files
- automatic application updates possible when signed by same private key

## Upon installation, the package manager creates a dynamic user ID for each application ⇒ **Application sandbox**

- all application files and processes are restricted to this UID
- enforced by Linux kernel and therefore same restrictions for all code (Java + native)
- by default, even the user and debugging shells are restricted to a special UID (SHELL)
- permissions granted at installation time allow to call services outside the application sandbox

**“rooting” to gain “root” access (super user / system level access on UNIX without further restrictions)**

# Android Security Architecture

## **Applications may install data on the device flash memory**

- /data/data/<package name> with sub-directories for files, databases, etc.
- API for centralized account data storage (one accounts database on system)
  - protected against normal applications
  - but no specific protection of this database on rooted devices (i.e. not encrypted)
- all files in application directories only accessible by the respective application UID

## **Applications may install (larger) data on extended memory (typically MicroSD)**

- however, MicroSD cards typically formatted with VFAT (or exFAT)
- ⇒ no DAC (Discretionary Access Control) by kernel, all files accessible to all apps

# Requesting Permissions

- For example, an application that needs to monitor incoming SMS messages would specify:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.app.myapp" >
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
    ...
</manifest>
```

- All requested permissions need to be granted at application installation (or otherwise, the application will not be installed) by explicit user consent
  - **How many users read (and understand) the list of permissions??**
  - Cyanogenmod  $\geq 7.1$  allows to selectively disable permissions for applications
- Permissions stored in `/data/system/packages.xml` are world readable, writable by user system (and root, obviously)
- Permissions checked (partially) at run-time by the respective services being called

# Demanding Permissions

- An application that wants to control who can start one of its activities could declare a permission for this operation:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.me.app.myapplication" >
    <permission android:name="com.myapplication.permission.DEADLY_ACTIVITY"
        android:label="@string/permlab_deadlyActivity"
        android:description="@string/permdesc_deadlyActivity"
        android:permissionGroup="android.permission-group.COST_MONEY"
        android:protectionLevel="dangerous" />
</manifest>
```

⇒ Allows user to decide if another application might call an activity.

- Declare activity as not being exported at all (can be launched only by components of the same application or applications with the same user ID)

```
<activity android:exported="false" ...>
```

- **Arbitrarily fine-grained permissions can be enforced with the `Context.checkCallingPermission()` method**

# Permission Checking

- **Activity permissions** (applied to the `<activity>` tag) restrict who can start the associated activity
- **Service permissions** (applied to the `<service>` tag) restrict who can start or bind to the associated service
- **BroadcastReceiver permissions** (applied to the `<receiver>` tag) restrict who can send broadcasts to the associated receiver.
- **ContentProvider permissions** (applied to the `<provider>` tag) restrict who can access the data in a ContentProvider. Unlike the other components, there are two separate permission attributes you can set: `android:readPermission` restricts who can read from the provider, and `android:writePermission` restricts who can write to it.

⇒ if the caller does not have the required permission then `SecurityException` is thrown from the call (some exceptions, e.g. BroadcastReceivers without the required permission will just not receive the broadcast)

# What can be done after rooting?

## Development

- Installing custom root, recovery, and system images
- [on devices without NAND lock] changing files on the `/system` partition

## General

- Reading all files on `/system` and `/data` partitions (including centralized system databases, e.g. accounts)
- Changing kernel settings, e.g. CPU over-/underclocking, IPv6 address privacy
- Custom routing, WiFi/3G connection sharing on devices that do not support “tethering” out of the box or have it disabled by the manufacturer/carrier, (re-)activating SIP, etc.
- Installing/using applications that require root access, e.g. Backup/restore, OpenVPN, Webkey, SSH daemon, etc.

# Rooting Overview

## 1. Achieve temporary root privileges

- on “developer” devices (Google ADP1 / G1, Nexus One, Nexus S) simply with
  - `fastboot oem unlock`
- booting into recovery mode on devices that allow this without restriction
- exploiting an implementation bug
  - either during normal system run-time
  - or in the boot loader code to get into recovery mode (see above)
    - e.g. flawed signature checking allows booting custom `recovery` image and from within that running image, flashing a new `system` image

## 2. Achieve permanent root privileges

- by flashing a new `system` image with pre-installed binaries for root access
- by installing a `/system/(x)bin/su` binary with “setuid” Bit set and the “SuperUser” application that will ask the user on each (first-time) access
- by modifying the `root` image, e.g. to set the global property `ro.secure=0`

## 3. Secure system against abuse by other applications

# Details: Linux kernel

## Android is based on Linux kernel

- “**root**” is the super-user, equivalent to system-level access
  - kernel implements DAC (Discretionary Access Control) on filesystem (includes kernel virtual files, e.g. /proc, /sys, etc.)
  - optional MAC (Mandatory Access Control) schemes available in upstream kernel
    - SELinux (NSA, flexible, comprehensive, but complex)
    - SMACK (simpler, path-based, used on MeeGo)
    - TOMOYO (path-based, more flexible than AppArmor)
    - AppArmor (simpler, path-based, used on Novell and Ubuntu servers)
  - could all be used to further restrict root user, often based on application context
- But none of them used in Android at this time!**

- Goal is therefore to achieve root privileges

# Details: filesystem ACLs

## Standard UNIX filesystem ACLs

- **read, write, execute** bits for owning **user, group**, and all **others**
- additional bits available, e.g. **setuid** and **setgid**
  - ⇒ binary called with privileges of owner, not privileges of caller
  - ⇒ typical combination is a file owned by root with setuid Bit set
- setuid binaries used on many UNIX/Linux systems to allow normal users to perform administrative tasks (e.g. `passwd`) or for arbitrary code elevation (e.g. `su`, `sudo`)
- for controlled root access, simple and effective method on Android
  - no changes to any existing system binaries
  - only need one additional binary installed with setuid Bit set
  - typically `/system/(x)bin/su` as on standard UNIX systems, but with Android-specific GUI to ask user for permission on root access

# Android Debug Bridge (adb)

- Must be enabled by user (USB Debugging)
  - but then available over USB, WiFi, or locally on device
- Supports debugging, file transfer, package installation, reboot control, etc.
- Normally runs as user SHELL (uid 2000)
- Can be restarted as user root (uid 0)
  - with global property `ro.secure=0`
  - then can call `adb root` to restart
  - or use one of the known exploits to force `adbd` to retain root privileges
  - will then support e.g. `adb remount` to remount `/system` with read-write option

# Devices with NAND-Lock

- Some HTC devices have a “NAND lock” implemented in their kernel and boot loader – also called “S-On” in contrast to “S-Off” (no additional protection)
- Boot loader sets global flag “S-On” for booting into Android and “S-Off” for booting into recovery mode
- NAND access is moderated by baseband (radio) processor, application processor has to go through it
- Radio firmware prevents write access to `system` (and sometimes `recovery`) NAND partitions even with root privileges and when `/system` has been re-mounted for read-write access
- Currently known devices with NAND Lock: all HTC Desire and EVO variants
  - HTC promised unlocking the bootloaders for all devices, but NAND lock??
- Other devices “only” have a boot loader that verifies signatures before installing updates for `recovery` and `system` and before booting a kernel (e.g. Motorola Milestone)



# Rooting: example exploits

## Exploits used for gaining temporary root privileges

- Android  $\leq$  1.6: missing input sanitization in udev firmware loading (CVE-2009-1185)
- Android  $\leq$  2.2: remapping shared memory (and system properties)
- Android  $\leq$  2.2: overflowing limit of processes for restricted SHELL user (`rageagainstthecage`)
- Android  $\leq$  2.3: restricting access to shared memory (`psneuter`)



<http://intrepidusgroup.com/insight/2010/09/android-root-source-code-looking-at-the-c-skills/>

# (Current) Conclusions on Android Security

## Android security measures are not sufficient

- **Recommendation 1:** ASLR, NX/XN exploit prevention; audit system level code
- **Recommendation 2:** more fine-grained application permissions (e.g. Internet)
- **Recommendation 3:** allow user to choose which permissions to grant each application instead of all-or-nothing at installation time  
⇒ Cyanogenmod  $\geq 7.1$  already does this
- **Recommendation 4:** MAC (kernel policies) in addition to DAC (filesystem ACLs) at least for critical binaries, better for application sandboxing (cf. SELinux application sandbox in current Fedora distribution)

# FH Hagenberg Project: Android Exploit FW

**Independent of specific exploit, as long as running code has (temporary) root privileges**

1. Remount `/system` for write access
2. Install new binary with setuid Bit set  $\Rightarrow$  e.g. "su" binary with SuperUser companion application
  - Exploit framework binary installs itself to `/system/bin` with setuid
  - **Alternative:** set setuid Bit on existing binary `/system/bin/sh`, possible due to bugs in YAFFS2 code in combination with NAND lock
3. Remove ACL restrictions from accounts and settings SQLite database files
4. Binary with setuid Bit is called at any future time when root privileges are required  $\Rightarrow$  permanent privilege escalation

# FH Hagenberg Project: Android Exploit FW

**Future work:** Automate process of creating custom NAND images to work around devices with NAND lock

1. Extract `boot` or `recovery` NAND partition to temporary files
2. For `boot` partition:
  - a) extract `initramfs` image
  - b) modify main boot script to set `ro.secure=0`
  - c) repack `initramfs` image
3. Write `boot` partition to NAND

Published as [S. Höebarth and R. Mayrhofer, "A framework for on-device privilege escalation exploit execution on android," in Proc. IWSSI/SPMU 2011, June 2011]

Framework + example exploits source online at <http://openuat.org/android-exploit-framework>

# FH Hagenberg Project: Android Encryption

## Aim: on-device encryption on Android devices

- `/data` partition should be fully and transparently encrypted, optional encryption of (micro-) SD card  
`/system` partition can remain unencrypted, as it only contains read-only Android device images (firmware)
- Based on in-kernel LUKS support with `cryptsetup` in `initramfs`
  - need to compare with upstream support when Android 4.x source is released
- Password either
  - entered by user during bootup
  - or stored on an NFC/RFID tag
- Interesting issues in system integration (UI is not available when `/data` is being mounted) and usability (entering a long password on an on-screen keyboard on each bootup is not fun!)

# FH Hagenberg Project: User Authentication

## **Aim: biometric user authentication on Android devices based on voice and face recognition**

- No biometric seems strong enough when used on its own
- Therefore combine face recognition (unobtrusive, quick, easy with front camera) with voice/speaker recognition when in doubt
- First prototypes working under Android, first evaluation promising
- Project will finish Q2/Q3 2012

Submitted for publication, source code will be available after project finishes

# FH Hagenberg Project: Pseudo-3D FaceRec

## **Aim: make face recognition more robust and more secure**

- Face recognition can (for most available systems) be easily fooled with a high-resolution image
- Idea: use more than one image with pseudo-3D face reconstruction
- Project just started, first results expected Q1/Q2 2012, will finish Q2/Q3 2013

# Most important aspect: **Usability**

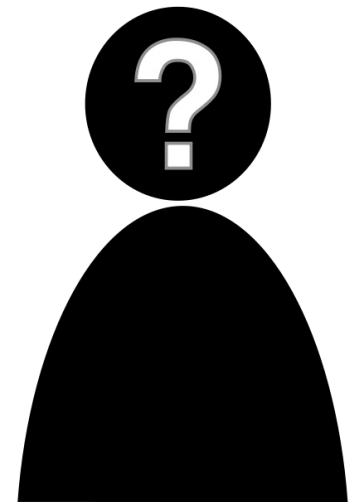
- When security methods or implications on users' privacy are not properly understood, systems will be used incorrectly
- Annoying and obtrusive security measures are simply deactivated so that users can get their jobs done
- For example:
  - sharing passwords, never logging out
  - writing PIN on back of card, most often used PINs "1234" and "0000"
  - "ALERT: The URL says www.mybank.com, but the certificate is for cracker.net, really continue?" - "Yeah, whatever, just let me enter my PIN and TAN codes now..."

When security and/or privacy and usability clash, then usability always wins!

# Secure mobile **usable** communication

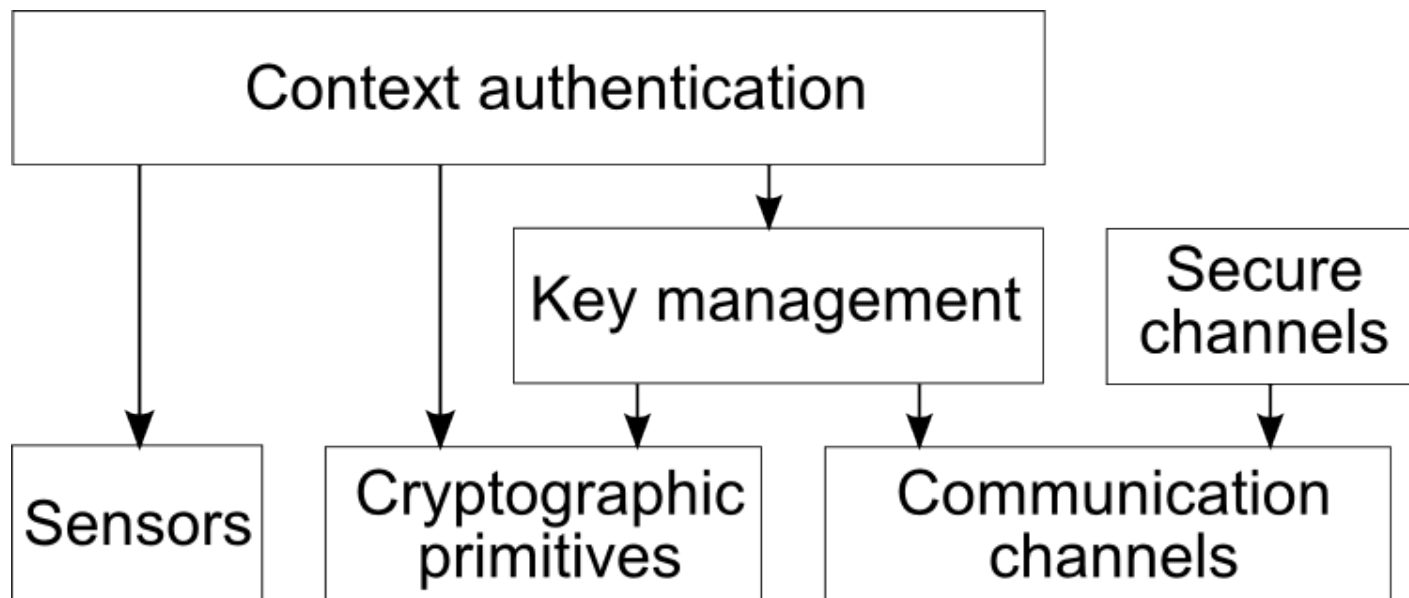
## Attention is a limited resource

- Vision of Pervasive Computing: daily use of **Hundreds** of services, ubiquitous and seamless embedding in daily life, **spontaneous** use, different administrative domains
- Continuous security popups, many different „good“ passwords **don't work in practice!**
- We need security measures that are  
⇒ **unobtrusive**, but **not invisible**



# OpenUAT

- Source code (Git), documentation, demo applications, data sets:  
<http://www.openuat.org>
- Mailing list, bug tracker: under construction (Redmine with Git backend)



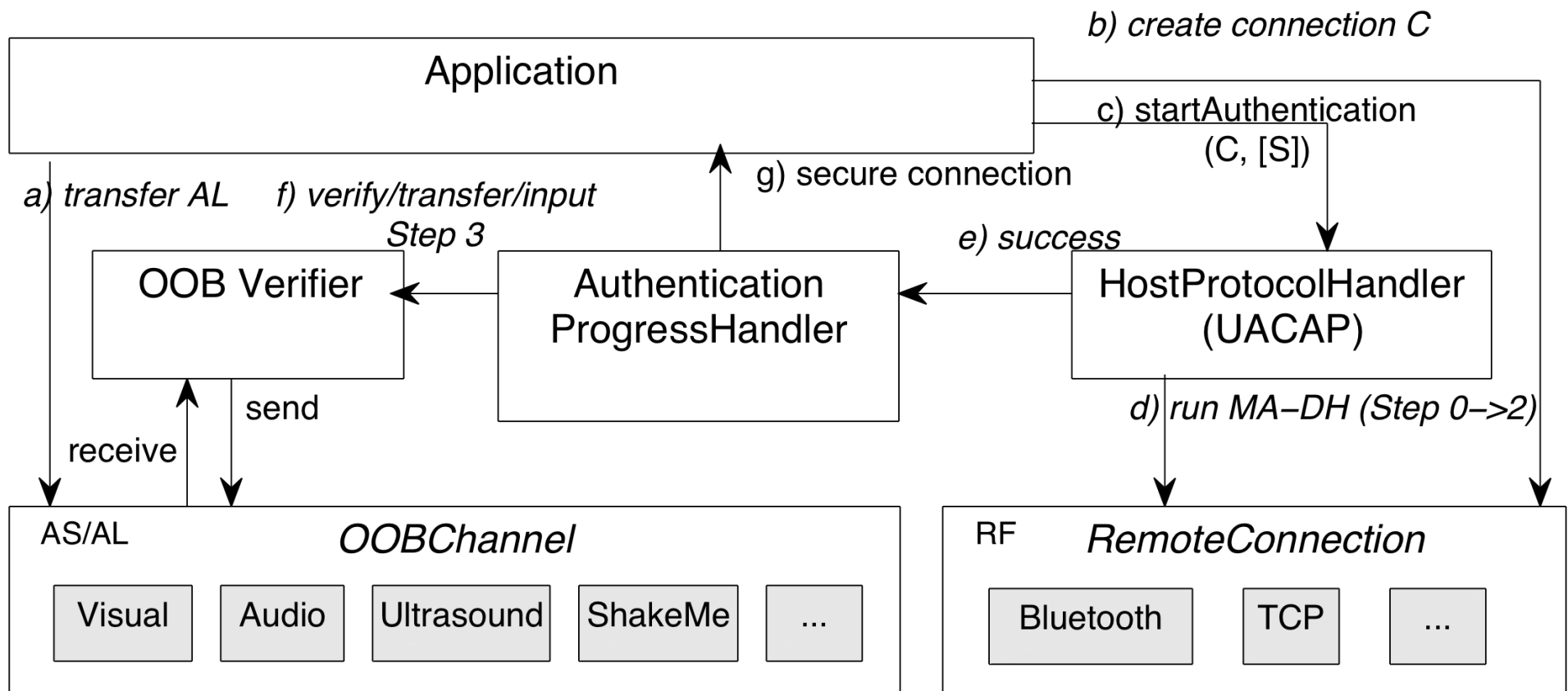
[R. Mayrhofer: "Towards an open source toolkit for ubiquitous device authentication", PerSec/PerCom 2007]

# Components in the current release

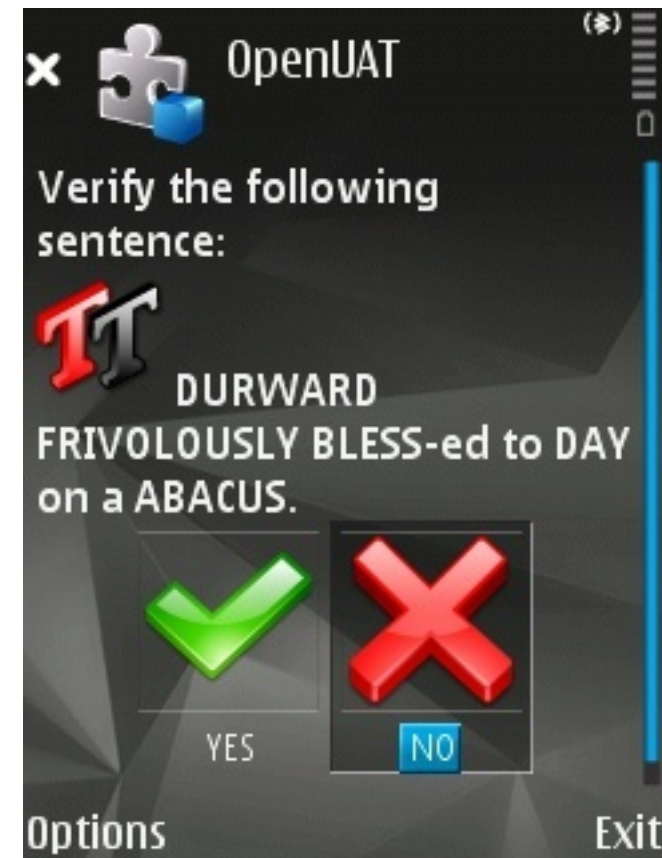
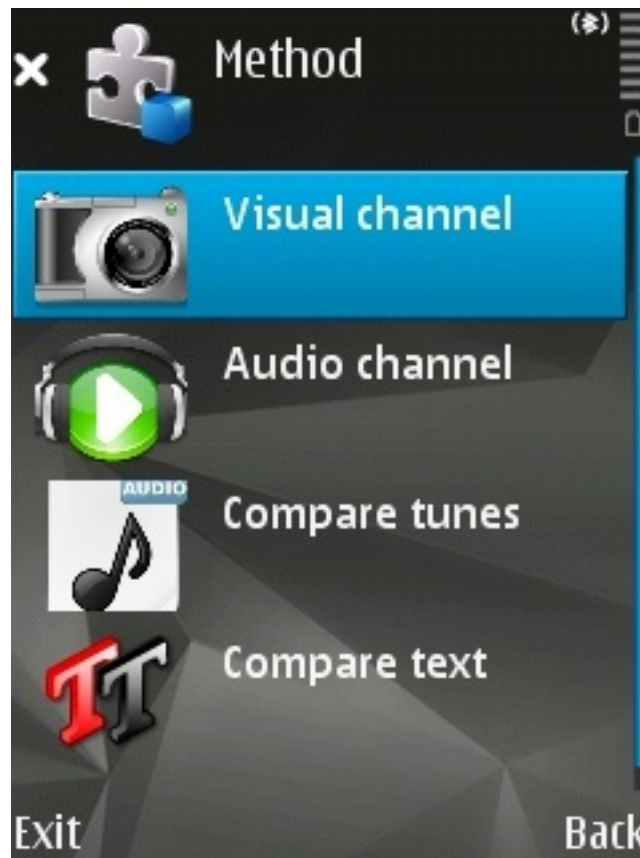
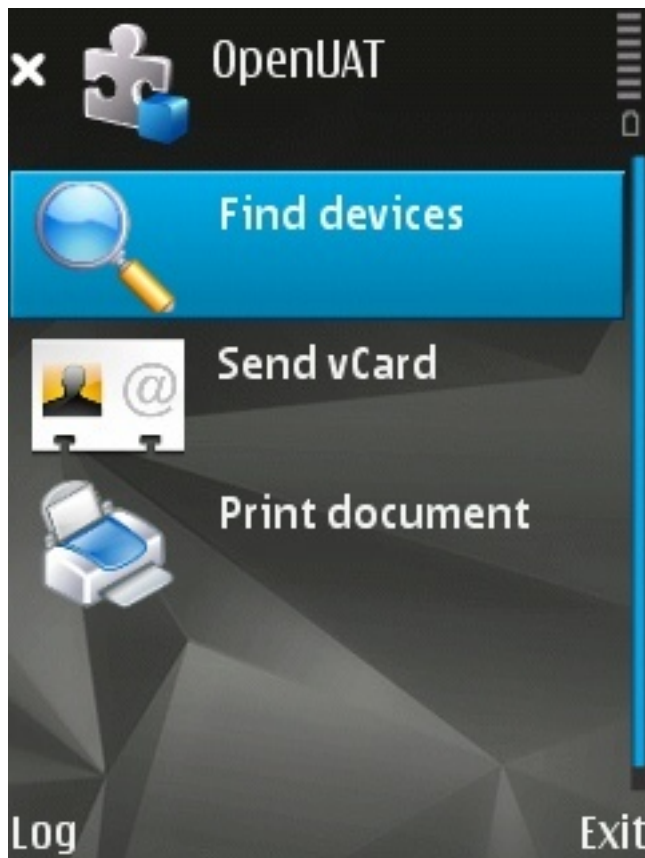
- **Cryptographic primitives:** ciphers, hashes (JCE and Bouncycastle with wrappers), DH with default parameters and utility methods, interlock\*, on-the-fly creation of X.509 CAs and certificates
- **Communication channels:** threaded TCP and Bluetooth RFCOMM servers using same interface (transparently interchangeable), UDP multicast, Bluetooth background discovery and peer management (opportunistic authentication)
- **Key management protocols:** DH-over-streams (TCP or RFCOMM), Candidate Key Protocol
- **Sensors and feature extractors:** ASCII line reader with various implementations for accelerometers, simple statistics, time series aggregation, activity detection/segmentation, FFT, quantizer
- **Context authentication protocols:**
  - Manual comparison
  - Spatial references
  - Visual: mobile phone camera + QR codes
  - Audio: midi tunes for transfer or verification
  - Accelerometer data
  - Simultaneous button presses
- **Secure channels:** IPSec tunnel and transport (Linux, MacOS/X, Windows), SSH and/or TLS under development

Utilizing Log4j/SLF4J, JUnit, Ant build system including J2ME builds

# Exchangeable in- and out-of-band channels



# Manual comparison of pseudo sentences



# Video Channel

- Workflow:
  - Hash encoded in a QR code
  - Take a picture of the other device and decode
- Experiences:
  - ZXing library for QR codes from Google
  - No lens autofocus support in J2ME for mobile phones, works much better under Android

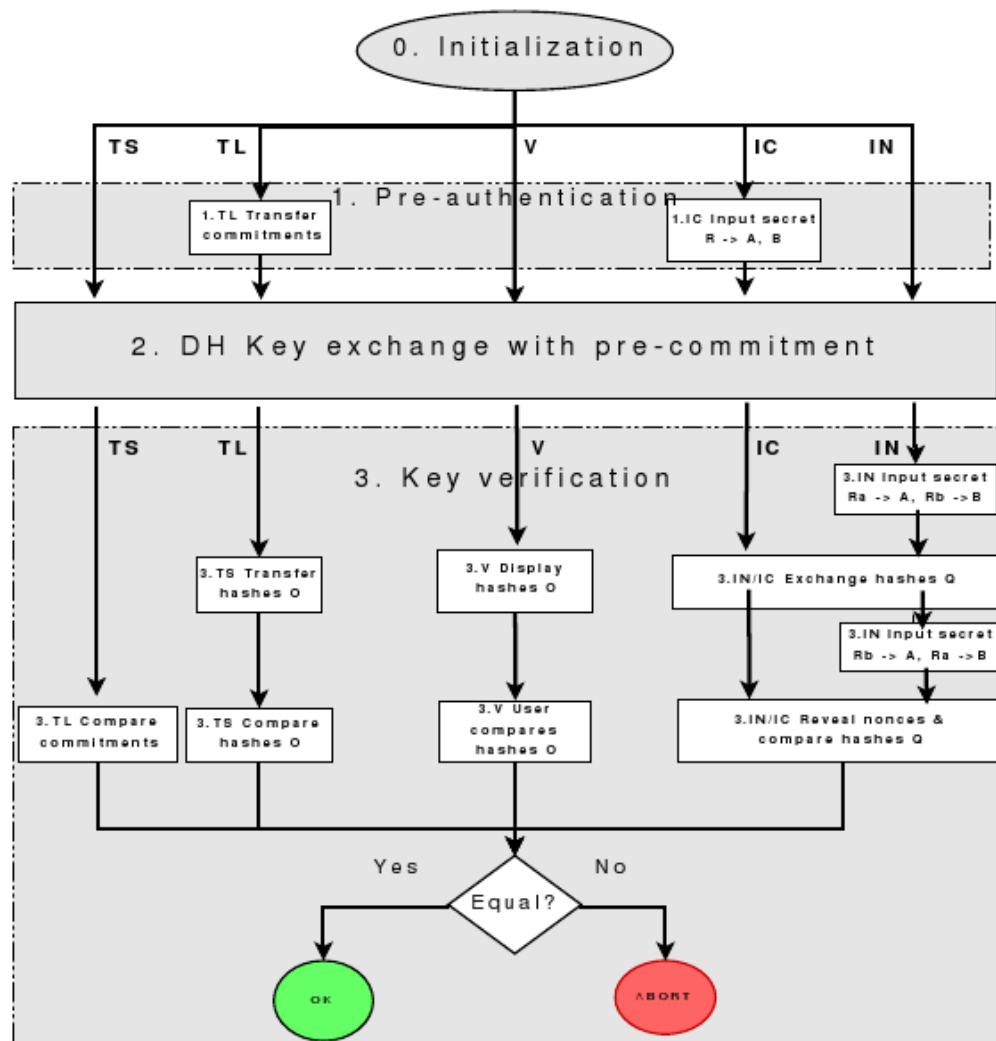


# UACAP

## Unified Auxiliary Channel Authentication Protocol

- Different modes of operation from user point of view
  - Input channels
    - IN: input non-confidential
    - IC: input confidential → pre-authentication possible
  - Transfer channels
    - TS: transfer "short" (10-60 Bits)
    - TL: transfer "long" ( $\geq 128$  Bits) → pre-auth. possible
  - Verification
    - V: explicit user verification (always "short"), non-confidential
- Main phases:
  1. Pre-authentication: only for TL and IC
  2. Diffie-Hellman key agreement with pre-commitment
  3. Out-of-band key verification

# UACAP: Unified Authentication Protocol



# UACAP Specification (1)

Channel	A (initiator)	Message	B (responder)
<b>0. INITIALIZE</b>			
	$I_a \in \{0 \dots 2^{128} - 1\}$ $x \in \{1 \dots q - 1\}$ $X := g^x$		$I_b \in \{0 \dots 2^{128} - 1\}$ $y \in \{1 \dots q - 1\}$ $Y := g^y$
<b>[1.TL: TRANSFER COMMITMENTS]</b>			
AL	$OC_X := Com(X)$ ( $\geq 128$ Bits)	$M_{0,a} := (I_a, OC_X)$ $\longrightarrow$	
AL		$M_{0,b} := (I_b, OC_Y)$ $\longleftarrow$	$OC_Y := Com(Y)$ ( $\geq 128$ Bits)
<b>[1.IC INPUT SECRET]</b>			
ACS	INPUT $R_a = R$		INPUT $R_b = R$
<b>2. KEY EXCHANGE (DH) WITH PRE-COMMITMENT</b>			
RF	$C_X := Com(X)$	$M_1 := (I_a, C_X[, P_a])$ $\longrightarrow$	
		$M_2 := (I_b, Y)$ $\longleftarrow$	
RF	$K_a := H(\tilde{Y}^x)$		
RF		$M_3 := (X)$ $\longrightarrow$	if $Com(\tilde{X}) \neq \tilde{C}_X$ then ABORT else OK else $K_b := H(\tilde{X}^y)$

# UACAP Specification (2)

## 3. OOB KEY VERIFICATION

$O I_a := (I_a   \tilde{I}_b)$ $O K_a := (X   Y)$ $O_a := \text{HMAC}_{O K_a}(O I_a)$ (ca. 10 . . . 60 Bits)	$O I_b := (\tilde{I}_a   I_b)$ $O K_b := (\tilde{X}   Y)$ $O_b := \text{HMAC}_{O K_b}(O I_b)$ (ca. 10 . . . 60 Bits)
---	---

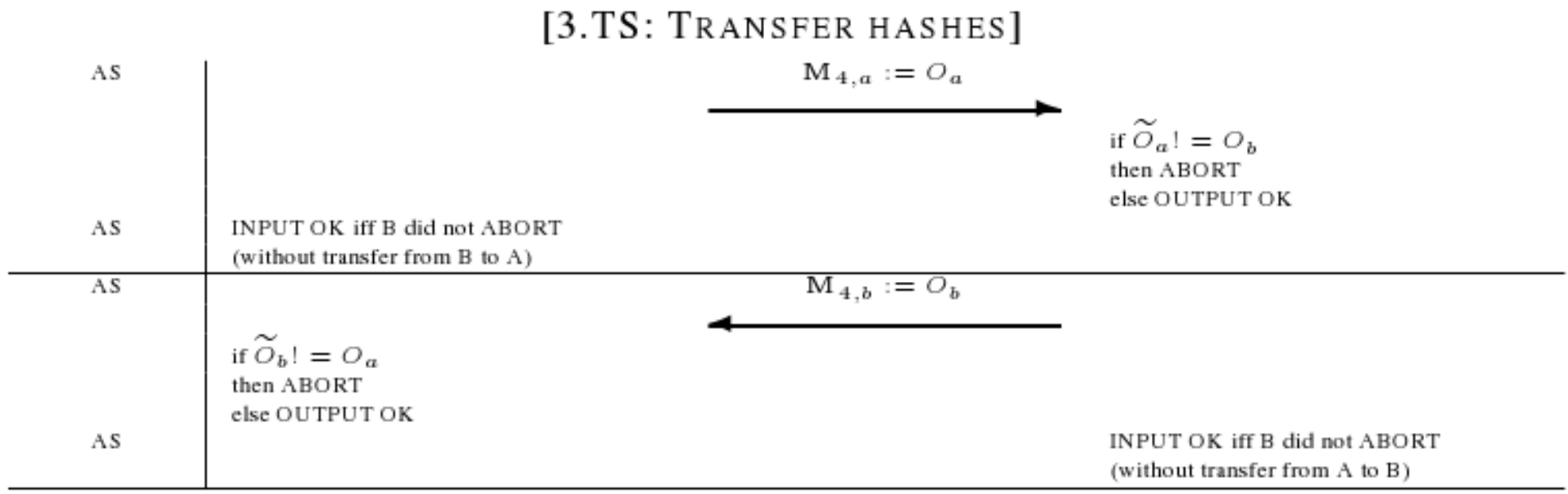
### [3.TL: COMPARE COMMITMENTS]

$Com(\tilde{Y}) \neq \tilde{OC}_Y$ then ABORT else OK	if $Com(\tilde{X}) \neq \tilde{OC}_X$ then ABORT else OK
--	---

### [3.V: DISPLAY HASHES $O$ , USER VERIFY]

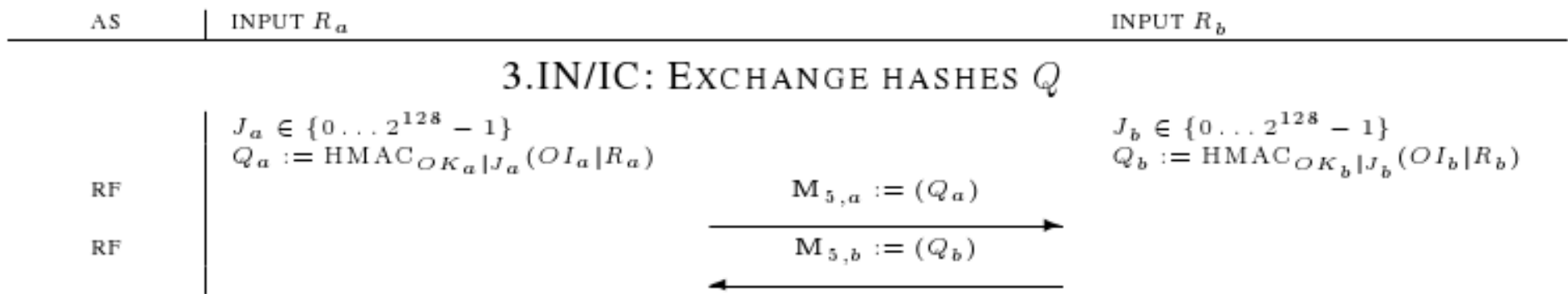
AS	OUTPUT $O_a$	OUTPUT $O_b$
AS	INPUT OK iff $O_a = O_b$	INPUT OK iff $O_a = O_b$

# UACAP Specification (3)

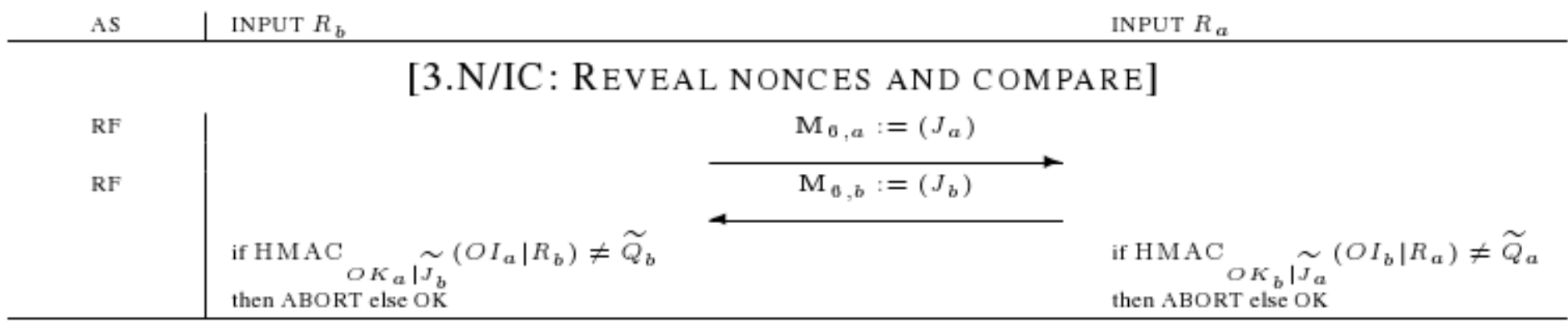


# UACAP Specification (4)

## [3.IN: NON-CONF. INPUT]



## [3.IN: NON-CONF. INPUT EXCHANGE]



# Using UACAP for Key Agreement

Creating a TCP socket to listen for incoming requests (**verify** or **transfer**):

```
HostServerBase server = new TCPPortServer(PORT, 10000, true, true);  
server.start();  
server.addAuthenticationProgressHandler(this);
```

Need to implement AuthenticationProgressHandler to listen for events:

```
public void AuthenticationFailure(Object sender, Object remote,  
    Exception e, String msg) {  
    System.out.println("Authentication FAILED with remote " + remote + ": " + msg);  
}  
  
public void AuthenticationProgress(Object sender, Object remote,  
    int cur, int max, String msg) {  
}  
  
public boolean AuthenticationStarted(Object sender, Object remote) {  
    System.out.println("Incoming authentication from remote " + remote);  
    return true;  
}  
  
public void AuthenticationSuccess(Object sender, Object remote,  
    Object result) {  
    System.out.println("Authentication SUCCESS with remote " + remote);  
  
    Object[] res = (Object[]) result;  
    sharedSessionKey = (byte[]) res[0];  
    byte[] shared00bMsg = (byte[]) res[1];  
  
    socketToRemote = (RemoteTCPConnection) remote;  
  
    // verify / transfer and verify shared00bMsg and, if equal, use sharedSessionKey  
}
```

# Using UACAP for Key Agreement

Connecting to a remote TCP socket and starting authentication

```
System.out.println("Connecting to remote " + connectTo);  
Socket client = new Socket(connectTo, PORT);  
HostProtocolHandler.startAuthenticationWith(  
    new RemoteTCPConnection(client),  
    this, 10000, true, "", true);
```

And **that's it** – both sides (client and server will receive their callbacks). Either use an implemented out-of-band channel or verify sharedOOBMsg in the application.

```
Hash.getHexString(sharedOOBMsg).substring(0, ...)
```

**Currently under development:** Android service for secure channel setup with out-of-band authentication, to be uploaded to Android Market with simple AIDL interface for arbitrary applications

# Research Laboratory on **Mobile Security**

## Open Issues in Research and Development

- **Basic research questions:** Hardware, cryptography, mobile operating system
- **Applied research:** Integration into platforms and enterprise structures
- **Social integration:** Better understanding by end-users

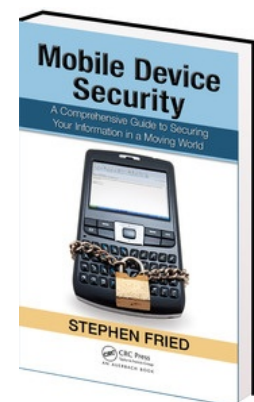
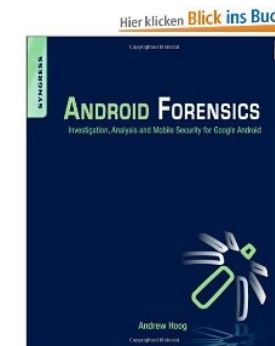
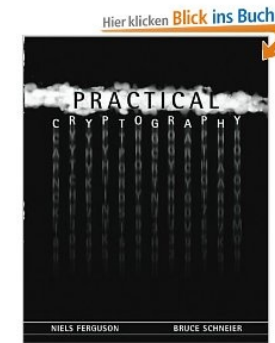
## Consortium

- **Laboratory lead:** Fachhochschule Hagenberg / JKU Linz
- Secure Business Austria
- NXP Semiconductors
- underground-8
- **And you?**



# Reading List

- Niels Ferguson and Bruce Schneier: „Practical Cryptography“, 2003
- Frank Stajano: „Security for Ubiquitous Computing“, Wiley Series in Communications Networking & Distributed Systems, Wiley, 2002
- Andrew Hoog: „Android Forensics: Investigation, Analysis and Mobile Security for Google Android“, Syngress Media, July 2011
- Stephen Fried: „Mobile Device Security: A Comprehensive Guide to Securing Your Information in a Moving World“, CRC Press, June 2010



# Thank you for your attention!

Slides: <http://www.mayrhofer.eu.org/presentations>

Later questions: [rene.mayrhofer@fh-hagenberg.at](mailto:rene.mayrhofer@fh-hagenberg.at)  
[rene@mayrhofer.eu.org](mailto:rene@mayrhofer.eu.org)

OpenPGP keys: 0x249BC034 (new) and 0xC3C24BDE (old)  
717A 033B BB45 A2B3 28CF B84B A1E5 2A7E 249B C034  
7FE4 0DB5 61EC C645 B2F1 C847 ABB4 8F0D C3C2 4BDE