



Secure Product Lifecycle

Penetration Testing for IoT Devices –
A Hardware Evaluator’s Perspective

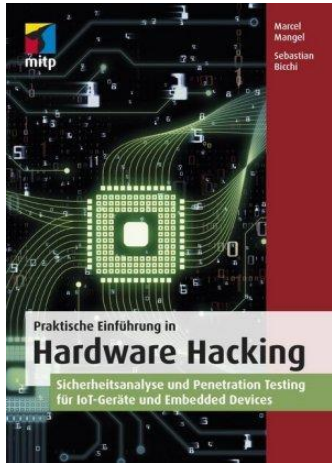
Christoph Herbst

WHEN YOU NEED TO BE SURE

SGS

- Security testing: Mobile Apps
 - Vulnerabilities and testing of mobile apps
 - White/Grey/Blackbox → we will revisit this part
 - Static analysis versus dynamic analysis
 - OWASP: MASVS (verification standard) and MASTG (testing guide)
 - OWASP Top 10 for mobile devices
 - Examples

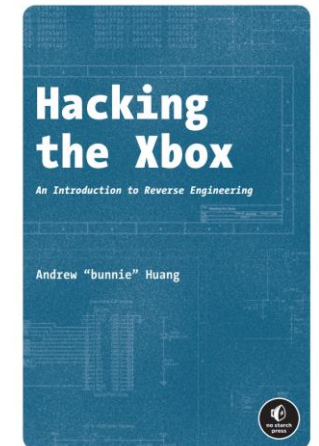
- IoT security is not just mobile (app) security and network/web penetration testing
 - This lecture focuses on IoT in general with a special focus on HW



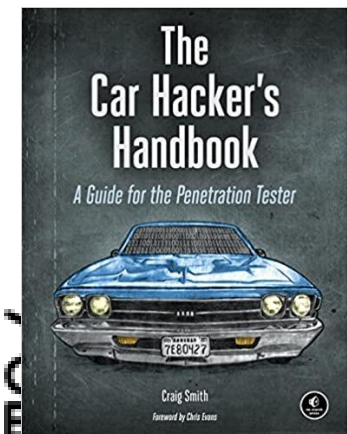
Marcel Küppers and Sebastian Bicchi, „Praktische Einführung in Hardware Hacking - Sicherheitsanalyse und Penetration Testing für IoT-Geräte und Embedded Devices“, MITP, Dec. 2019

Andrew Huang, „Hacking the Xbox - An Introduction to Reverse Engineering“, No Starch Press, Inc., 2003

<https://nostarch.com/xboxfree>



Craig Smith, „The Car Hacker's Handbook: A Guide for the Penetration Tester“, No Starch Press, Inc., 2016



EXAMPLES OF IOT HACKS



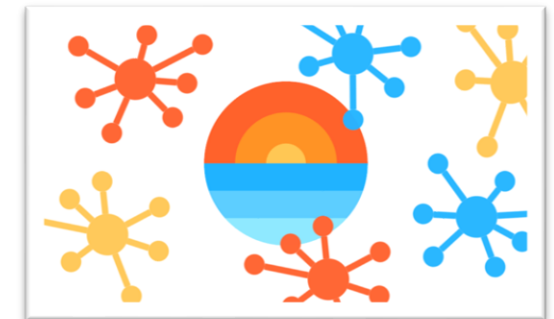
Insurance company refuses to pay after car hack via „Keyless Go“ system
→ court confirms rightfulness (2020)



“Hackers Remotely Kill a Jeep on the Highway” (2015)



„Hello Barbie“ – Insufficient security allows hackers to: eavesdrop & speak over the Internet (2015)



Kaiji Botnet uses automated attacks against SSH to infect IoT devices (2020)



“Hackers are hijacking smart building access systems to launch DDoS attacks” (2020)



WHITEBOX VS. BLACKBOX TESTING

■ Blackbox

- Tester gets same (public) information as a consumer
- Unknown algorithm details and countermeasures
- Samples same/similar to product
- No further support from developer

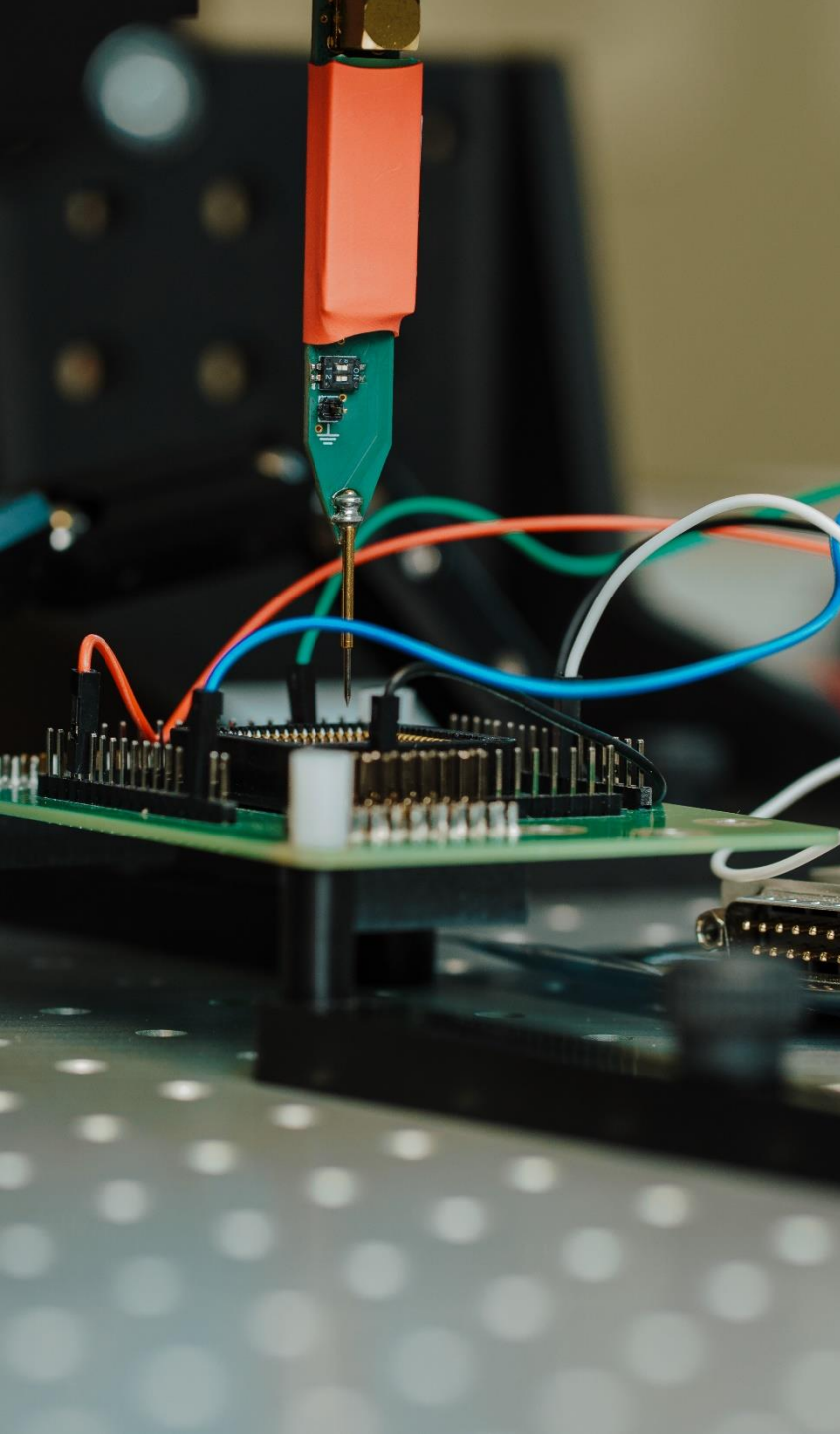


GREY BOX

■ Whitebox

- Tester has in-depth knowledge about internals
- Algorithms and countermeasures declared and described in detail
- Samples prepared for testing (opened, added functionality)
- Support from developer
- → More common in high-security domains

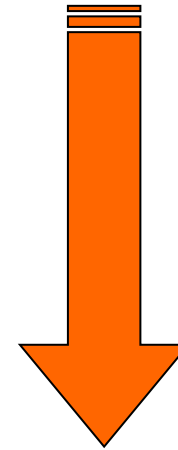




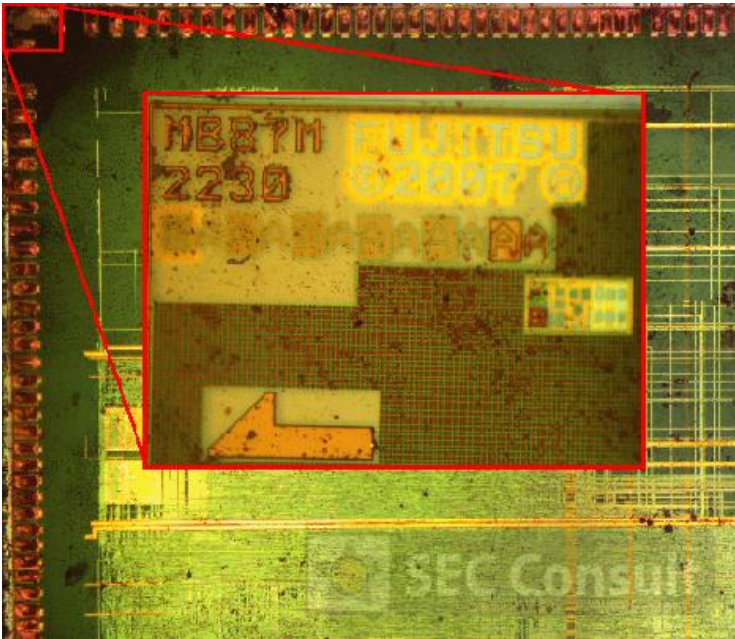
IoT Hardware Evaluation Flow

IOT HARDWARE ANALYSIS FLOW

1. Identification of samples
2. OSINT analysis
3. Remove casing
4. Component identification
5. Penetration tests
 - a) Security review PCB design
 - b) Debug interfaces & configuration ports
 - c) Memory interfaces
 - d) Firmware and configuration corruption
 - e) Glitching of security countermeasures
 - f) Probing of data buses
 - g) Interfaces tests (secure comm., analysis, attacks)
... up to dedicated device tests



1. IDENTIFICATION OF SAMPLES



<https://sec-consult.com/en/blog/2019/02/reverse-engineering-architecture-pinout-plc/>

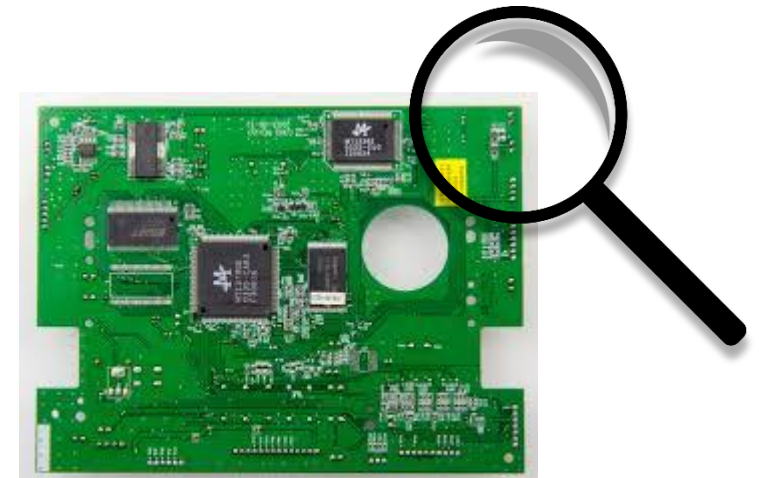
→ Interesting article



- Check that what you are trying to test is actually the TOE that is intended to be tested!
- Depends on device that you test (e.g. label on device, or device output at start-up → identification and version)
- Sounds trivial but if you perform the evaluation on the wrong device there is lots of work done without any use

2. OPEN SOURCE INTELLIGENCE ANALYSIS (OSINT)

- Exploring the publicly available data sources
 - Data sheets and user manuals
 - Firmware if available
 - Reports and blogs
 - Papers, etc.
 - Specific search engines
- Information gathered
 - Use cases and technologies
 - Interface descriptions
 - Hardware structure & components
 - Data flow diagrams
 - Known attacks & standard passwords
- → Make the black box a bit more greyish



3. REMOVE CASING – GETTING INTO THE DEVICE

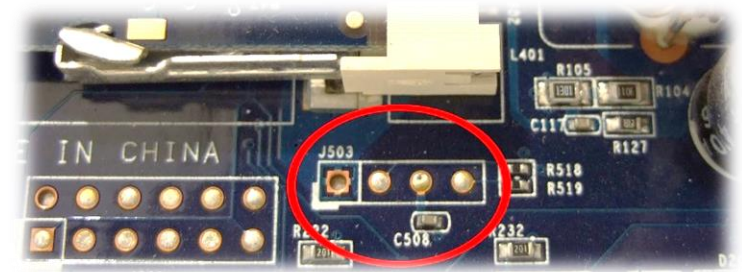
- Can be very trivial or very complex
- Sometimes considered as part of the physical security of the device
 - Limited testing time!
- Destruction not always an option
 - Limited samples
 - Some tests/attacks required an open and working sample
- Goal: get to the electronic parts



4. COMPONENT IDENTIFICATION



- Starting point: limited information on internal electronic parts
- Denote all main electronic components
 - SoCs
 - CPUs, μ Cs
 - Memory ICs
 - Debug & config ports
 - Disconnected interfaces →
 - Everything that could be security relevant
e.g. suspicious or undocumented interface
- → Back to OSINT (update information)
- Output: list of components with an assessment of possible security impact



5. PENETRATION TESTS

- Do whatever an attacker does...
- Inputs
 - Information from Component Identification
 - OSINT
- Create a test plan
- Keep in mind for planning:
 - Scope (what was agreed on with customer?)
 - Limited available time (up to this point already consumed >2-3 days)
 - HW pentests are not the only tests to be run (network pen testing, mobile app tests)
 - Attack potential for IoT (basic to substantial)
 - What tests are the most critical ones (low-hanging fruits for an attacker)?
 - Hardware reverse engineering and bespoke equipment is usually out of scope
 - Coverage



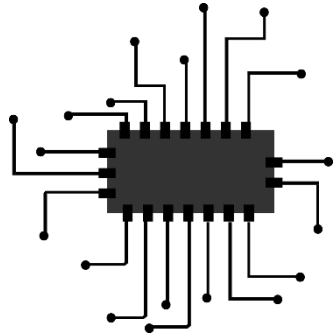
Hardware Pentesting Examples



RoT



- What is a Root of Trust?
- Foundation for all security related functions
 - e.g. Boot Loader relies on RoT for loading firmware image
- Examples
 - Hardcoded credentials (passwords) → not a good idea
 - Hardware security module (HSM)
 - Secure memory (one-time writable, limited access)
 - Physically unclonable functions
- Assume to be secure during evaluation
 - verify it, if possible
- Often hardware security modules seen as RoT
 - trusted execution environment (TEE) → actually not a module
 - execution of cryptographic functions (signature, de- and encryption)

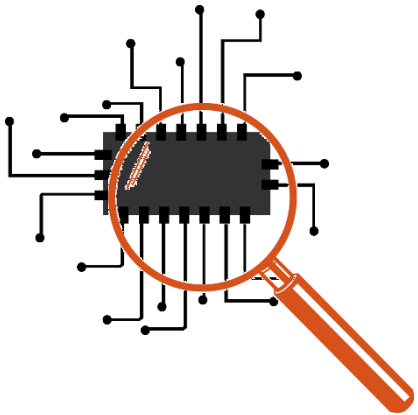


- Threat model/scope
 - Physical attacks in scope?
- Tamper resistance
 - how is it achieved?
 - E.g. on-chip voltage regulators, attack sensors, redundancy, error counters...
- Security perimeter
 - What needs to be protected? What relies on the RoT?
 - Primary assets (e.g. user data, firmware, IPs)
 - Secondary assets (e.g. cryptographic keys)
 - Countermeasures & components rely on RoT
- (True) random number generators
 - Lots of countermeasures and cryptographic functions depend on (T)RNG
 - Different classes: PTRG.1-3, DRG.1-4 → different use cases
 - Check soundness through online tests



HOW TO EVALUATE „ROOT OF TRUST“?

- Check that assumptions on RoT are meaningful & consistent
 - e.g. External flash assumed to be secure memory for static passwords
→ not a good idea ☹
 - RNG class suitable
- Check for tamper resistance if in scope
 - E.g. glitching, underpowering
- Usually no direct tests on RoT
 - → no interface provided to test assumptions (cf. CC evaluations)





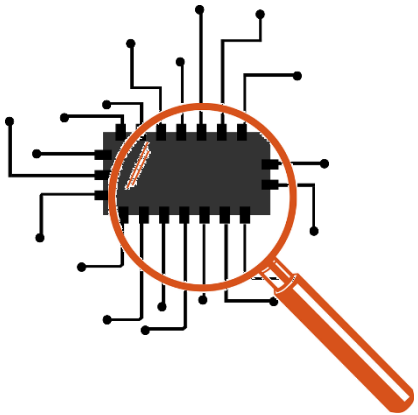
- Security feature of many embedded devices
 - Integrity & authenticity of code and data stored in non-volatile storage
 - Often replay/downgrading protection
 - Bring the system to a defined & secure state after start-up
 - → if secure boot can be bypassed the whole system can be compromised
- Implemented on a wide variety of devices
 - E.g. phones, TVs, automotive, routers, consoles etc.
- Hard to analyse and often unexplored
 - Complex and tight hardware & software interaction
 - High privileged functionality implemented at the lower levels
- All Secure boot implementations are
 - different but lots of similarities



- Became a common chip feature
- Requires hardware to be secure (RoT)
- Relies on (strong) cryptographic primitives
- Must verify all code and data
 - To bring system into a defined secure state
- Breaking Secure Boot early usually grants higher privileges
 - Key, ROM code, Countermeasures, HW protection, Code execution

HOW TO TEST THE BOOTLOADER?

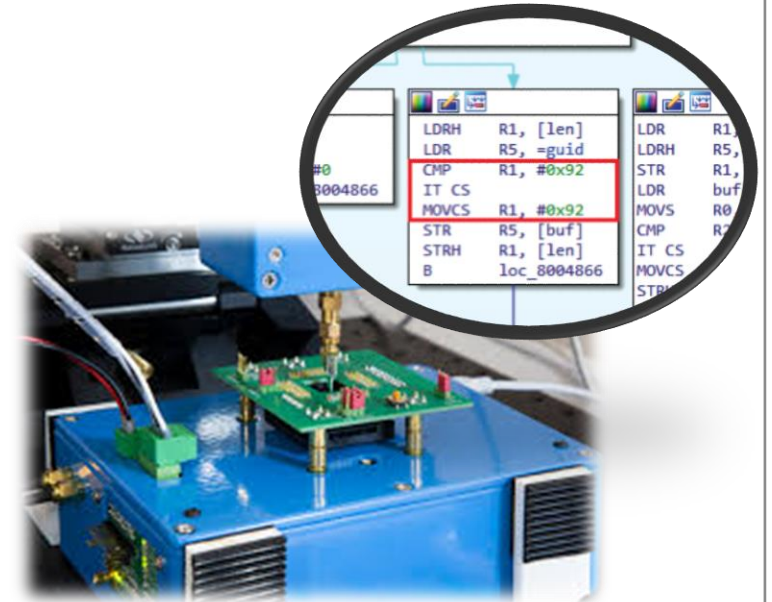
- Change firmware image → see if bootloader reacts at all
- Perform glitching attacks (next slide)
- Try other ways to overcome bootloader
 - E.g. TOCTOU (time of check/time of use) attacks
 - → Change flash image after check
- Full key extraction of NVIDIA TSEC:
<https://gist.githubusercontent.com/plutooo/733318dbb57166d203c10d12f6c24e06/raw/15c5b2612ab62998243ce5e7877496466cabb77f/tsec.txt>



GLITCHING OF SECURITY COUNTERMEASURES

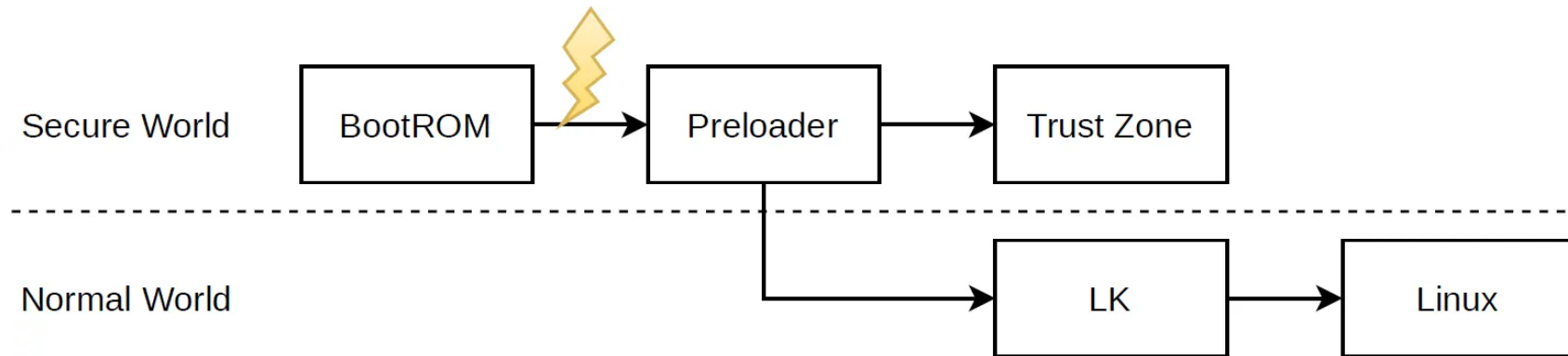
- Different ways to glitch a device
 - Changing environmental conditions (heat/cool)
 - Flashlight (cheap & powerful EM source)
 - Clock glitching
 - (Supply) voltage glitching
 - EMFI/BBI glitching

- Goals
 - Bypass a security check → e.g. Bootloader
 - Bypass authentication → e.g. Login
 - Fault analysis



EXAMPLE ATTACK AGAINST THE BOOTLOADER

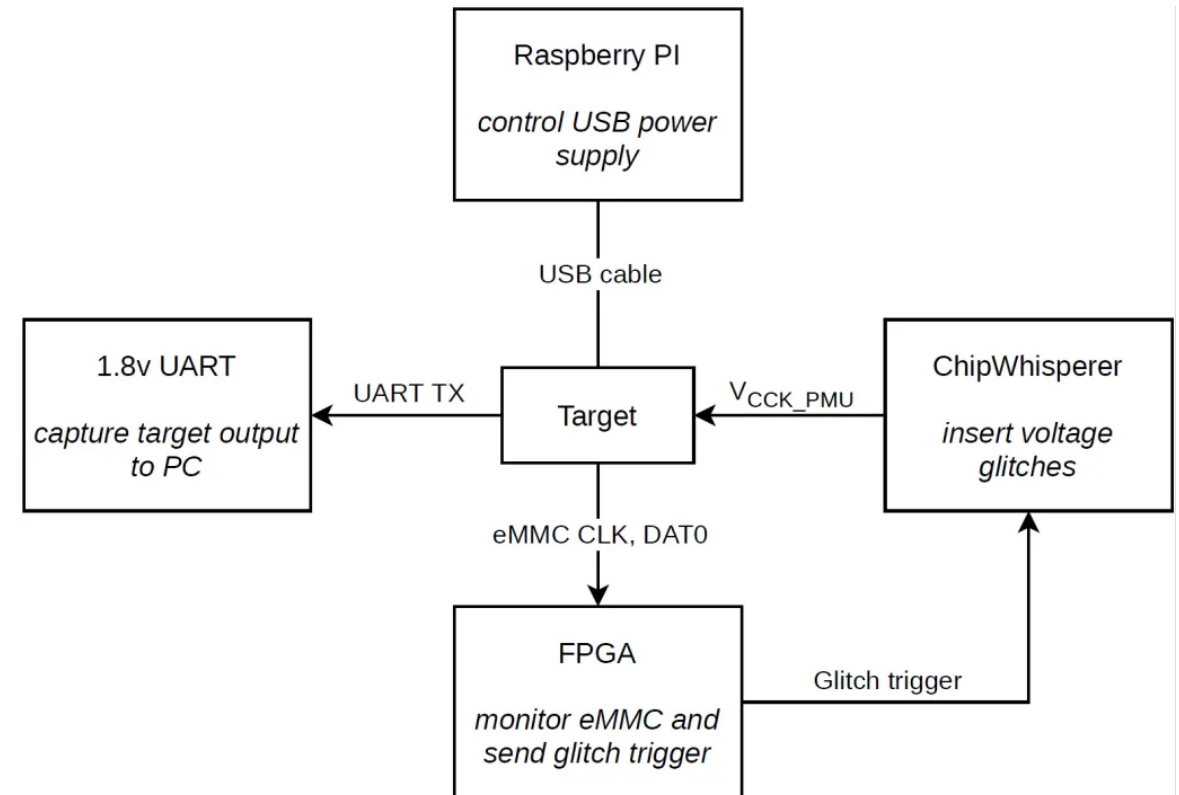
- Attack on MediaTek MT8163V system-on-chip
 - used in Android tablets



- source: <https://research.nccgroup.com/2020/10/15/theres-a-hole-in-your-soc-glitching-the-mediatek-bootrom/>

EXAMPLE ATTACK AGAINST THE BOOTLOADER

- UART for behaviour monitoring (valid boot image versus invalid boot image)
- Timing information
- FPGA for precise timing of voltage glitching
- Chip Whisperer for voltage glitching of bootloader
- Hardware costs below €500



EXAMPLE ATTACK AGAINST THE BOOTLOADER

- Timing analysis of boot process
 - → successful vs unsuccessful boot
- Verification takes about 700ms
 - time frame for glitching
 - start time calculated via last bytes of bootloader image
 - exact glitch timing brute forced
 - verification via change of debug string
 - in the end, success rate of 25% reached



TRUSTED EXECUTION ENVIRONMENTS

- What most people believe it is:
 - „Processor feature“ or „Secure place in your hardware“ (like HSM/Secure Element)

- TEE is...
 - Not a specific component, but multiple hardware & software components
 - Not a feature that can be switched on or off
 - Typically not made by a single manufacturer

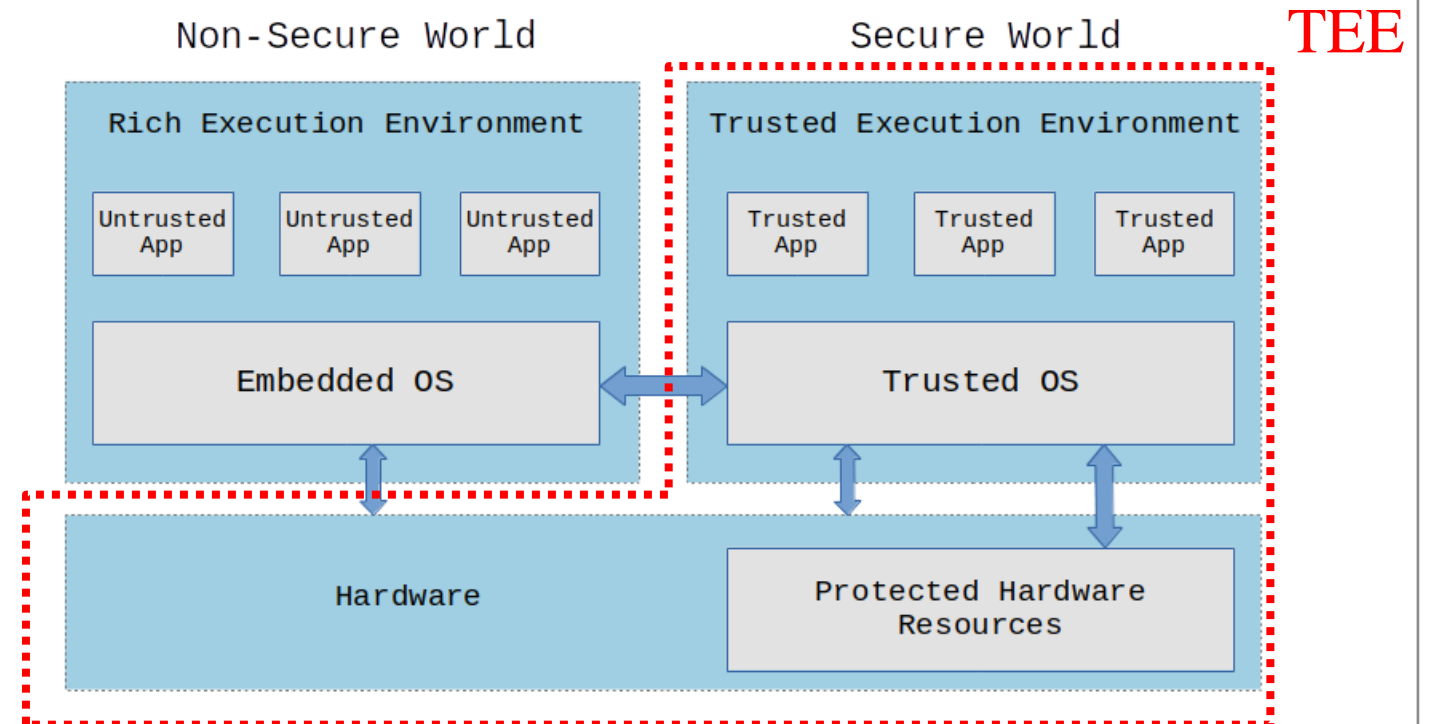
- Purpose:
 - Protect TEE world from the Normal world (REE world)
 - Protect trusted applications (TAs) from
 - REE & other TAs

TrustZone[®]
System Security by ARM



TRUSTED EXECUTION ENVIRONMENTS

- Example ARMs TrustZone
- Software adds flexibility (e.g. Update in the field)
- Hardware enforces separation
 - → NS bit is everything that hinders access to TEE memory
 - Can only be set in S-EL3 “Monitor Mode”
- Communication over SMC calls (not in user mode/EL0)



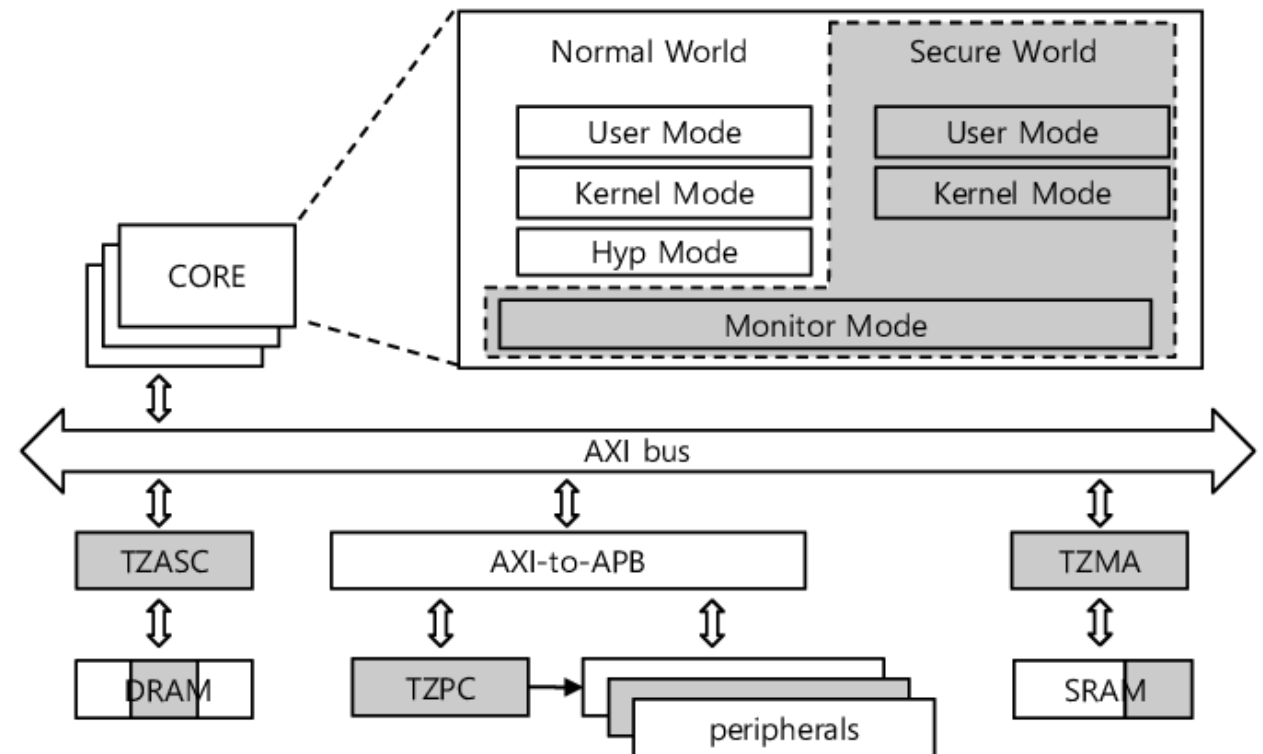
<https://embeddedbits.org/introduction-to-trusted-execution-environment-tee-arm-trustzone/>

TEE – SEPARATION VIA THE NS-BIT

- Dedicated hardware components enforce access policy
 - TZASC... TZ Address Space Controller
 - TZPC... TZ peripheral controller
 - TZMA... TZ Memory Adapter

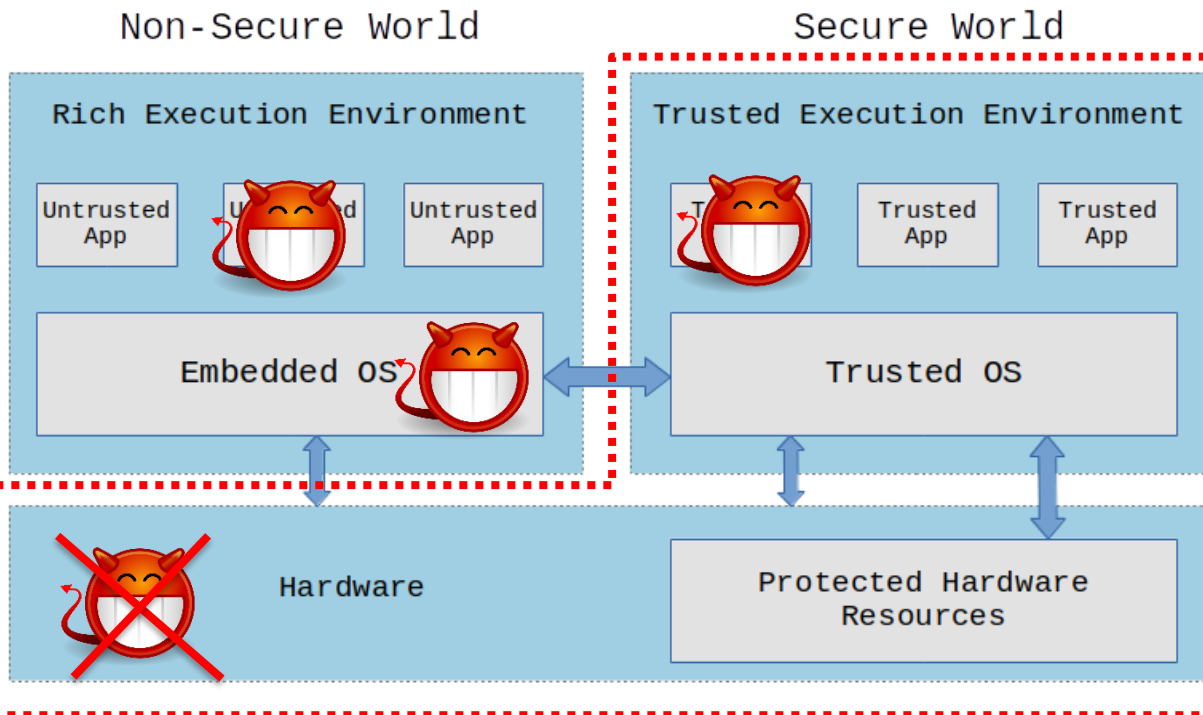
- Check if NS-bit=0 to grant access to certain hardware parts

- Pitfall: misconfiguration of access table in hardware and TEE monitor software (needs to be the same) → Otherwise REE can let TEE read/write secure memory

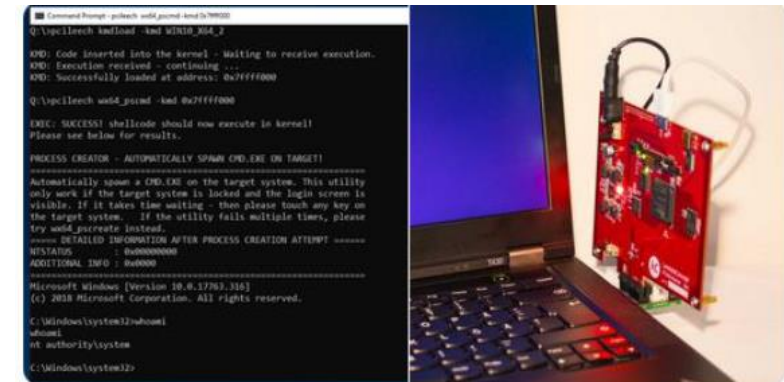


https://www.researchgate.net/figure/Components-of-ARM-TrustZone_fig1_304123847

- REE to
 - HW, TEE OS, TAs
- TA to
 - HW, TEE OS, Other TAs
- Physical attacks, e.g. DMA attack



<https://embeddedbits.org/introduction-to-trusted-execution-environment-tee-arm-trustzone/>



→ Not in scope of TEE threat model

HOW TO EVALUATE THE TEE?

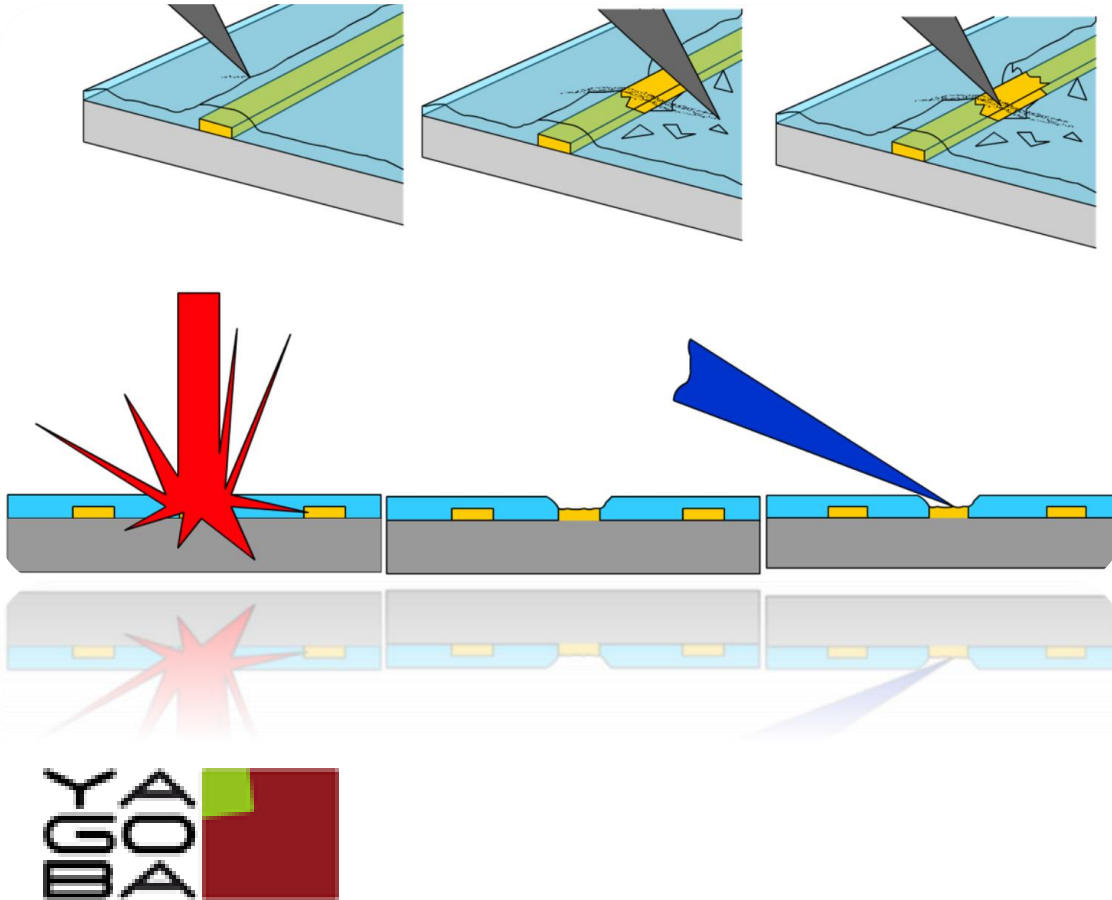
- Check configuration documentation (HW vs. SW)
- Check for known exploits (OSINT)
- Fuzzing of TA/Trusted OS functionality
- If interfaces are available:
 - Write dedicated test code (TA)
 - Dedicated tests targeting TA functionality



Physical Security



PROBING OF DATA BUSES



- Eavesdropping on internal communication (e.g. CPU ↔ memory)
- Scratch the surface with a needle or knife and contact it
- Also possible on IC die-level but out of scope for IoT
- Goals
 - Extraction of unencrypted firmware during booting
 - Read keys or configuration
 - Insertion of faults

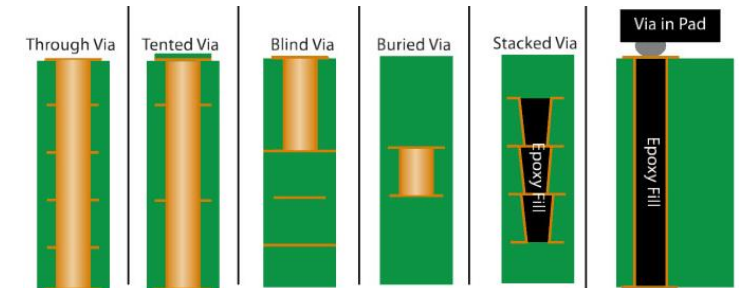
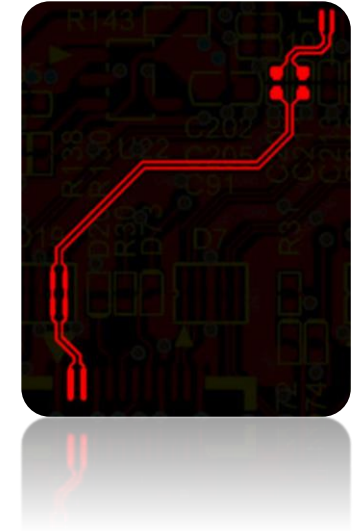
SECURITY REVIEW PCB DESIGN

- Many of the (potential) weaknesses, security vulnerabilities, and design flaws of a product are identified when reviewing the PCB
- How difficult to access the PCB, busses and components?
 - More difficult means more time and/or more samples
- Physical access
 - Blocked PCB physical access with secure mechanical cover
 - Compromise detection used (open case detection, seal the PCB)
- → not exactly Kerckhoffs's principle



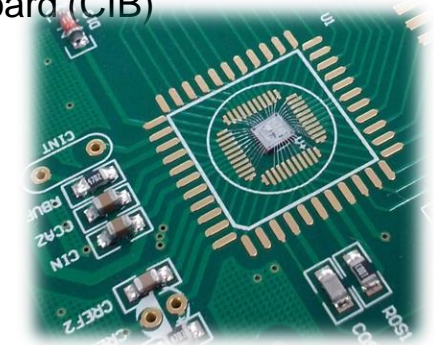
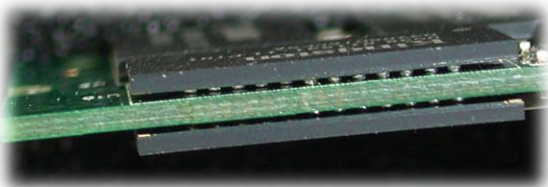
PCB design techniques/rules

- Dense PCB design
- Power lines have buffers (capacitors) → place as close as possible to ICs
- Wires
 - place as shortly and directly as possible
 - Critical traces placed in the inner layers
 - Differential lines designed parallel (cf. figure, also in case if they are on separate layers)
- (multiple-) Ground layers sandwiched for the sensitive traces with EM radiation (EMI shielding)
- Vias
 - Buried and/or blind vias used when it is possible →
 - Buried via with nonconductive via fill →
- Unnecessary test points and programming headers removed
- Fake test points, components placed
- One-time solder paste used



■ Components

- IC-s (μ C, memory etc.) with enabled security features (secure boot, authentication etc.)
- Components which have limited availability access to the datasheet
 - Custom design IC-s
- Black out chip label or IC markings removed or changed
- Physical access
 - Advantageous packaging
 - Ball Grid Array (BGA) packaging, Chip-on-Board (COB)
 - Embed IC-s inside the PCB, (layer stackup) applied, Chip-In-Board (CIB)
 - Epoxy/molding mass around package
 - Embedded planar capacitor used (buried capacitors)



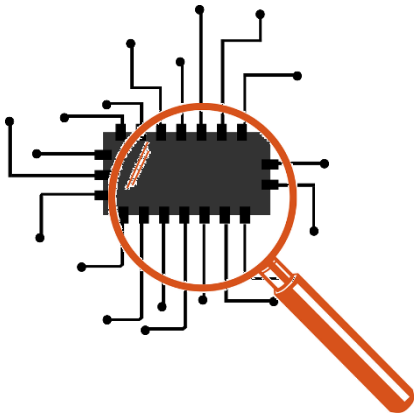
HOW TO EVALUATE THE PCB DESIGN?

- Check design rule suggestions

- Grade how difficult it is to...

- Open
- Identify
- Probe
- Remove

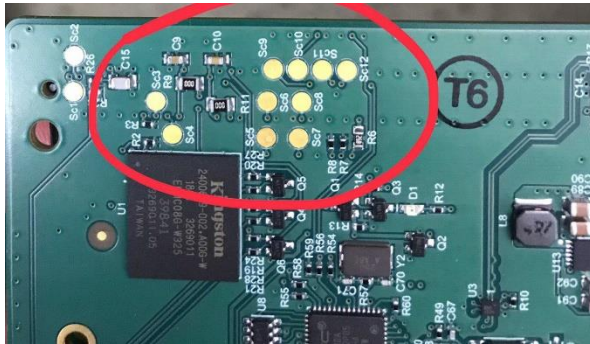
...certain components



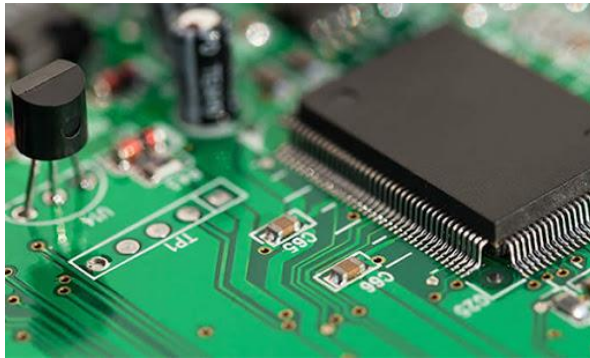
Interfaces & Memories



DEBUG INTERFACES & CONFIGURATION PORTS



Spring contact test points



Unsoldered pins

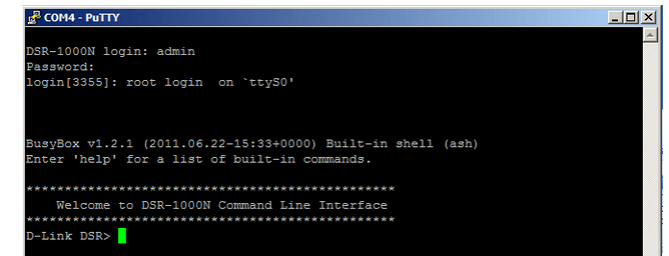
Design-for-**Security** vs. Design-for-**Testability**

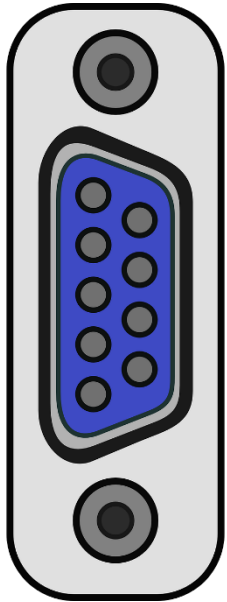


- PCB test points
- Unsoldered pins
 - Config ports/management ports
 - Debugging interfaces
 - UART, SPI, JTAG,...
 - Open debug ports
- What can you do with it?
- Read/write internal state (e.g. keys)
- Basically controlling the device
- Strategy considered to permanently remove test functionality?

Configuration ports

- Might get a CLI →
- Standard passwords?
- More privileges/functions

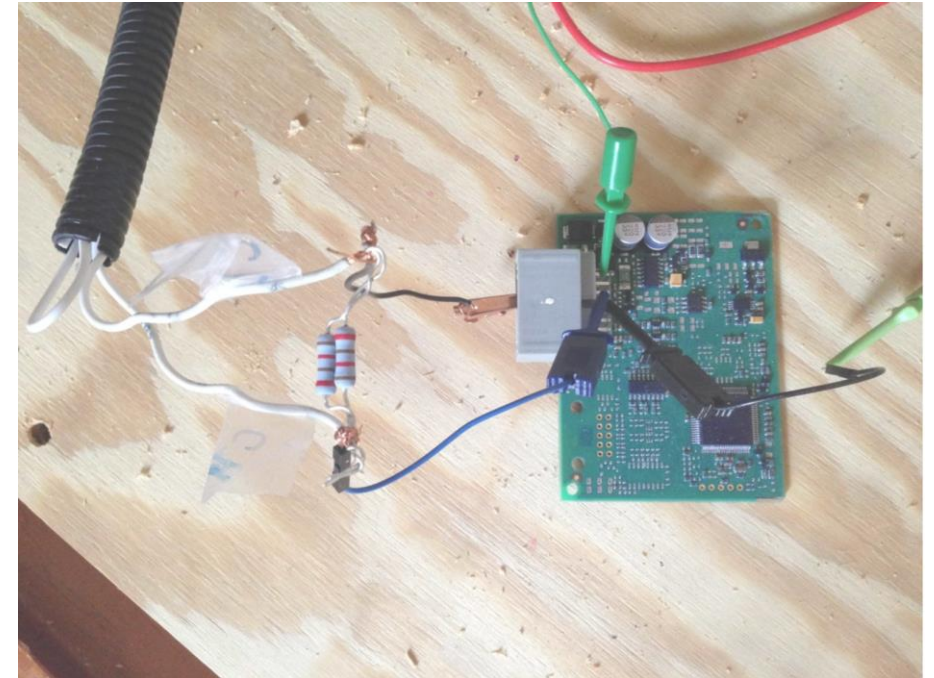




- Interface complexity can differ highly
- Example automotive:
 - CAN bus
 - easy to access, 2 resistors and lines to connect to
 - simple electronics to retrieve and send signals
 - FlexRay
 - time sensitive bus system
 - special hardware to connect to
 - equipment usually FPGA based, and expensive
 - configuration details required to properly decode messages
- Complete interface analysis required for additional attacks
 - fuzzing

INTERFACE ANALYSIS – CAN BUS EXAMPLE

- Access requires only two 120 Ohm resistors
- Once the ECU is stable, one can analyse the traffic
 - getting to the stable state is hard
- CAN messages are ID + max. 8 bytes



http://illmatics.com/car_hacking_poories.pdf

```
IDH: 00, IDL: 81, Len: 08, Data: F9 5C 01 00 00 00 00 00
IDH: 00, IDL: 81, Len: 08, Data: F9 5C 01 00 00 00 00 00
IDH: 00, IDL: 81, Len: 08, Data: F9 5C 01 00 00 00 00 00
IDH: 00, IDL: 81, Len: 08, Data: F9 5C 01 00 00 00 00 00
```

http://illmatics.com/car_hacking_poories.pdf

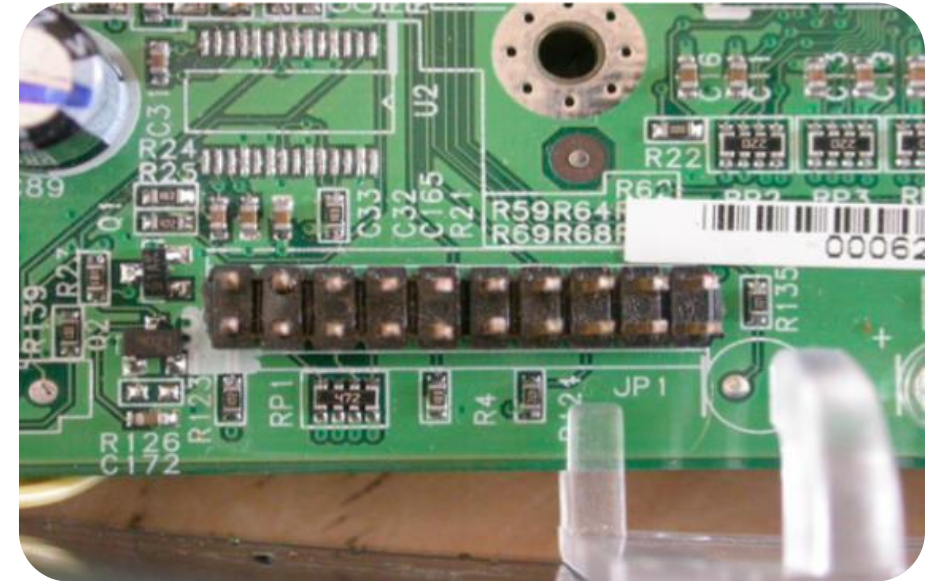
CONFIGURATION INTERFACE EXAMPLE - UART

- Simple serial interface
 - 2 lines (RX/TX) + VDD +GND
- Baud rate requires configuration
- Simple connectors available
 - UART to USB
- How to determine pin layout?
 - VDD/GND → shape of pins
 - brute force



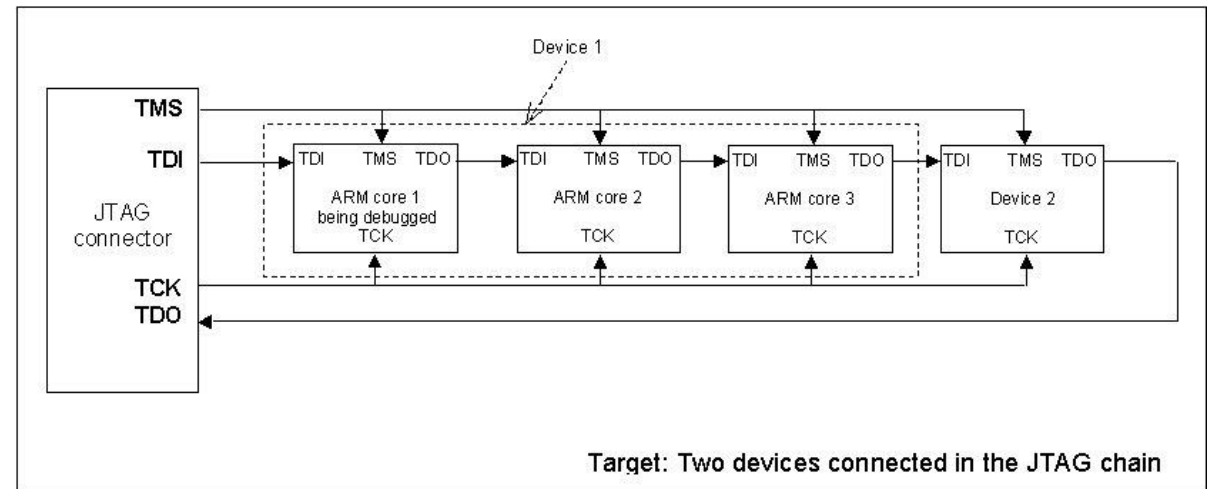
<http://www.devtty0.com/2012/11/reverse-engineering-serial-ports/>

- Officially 4 + 1 pins
- no official protocol
- no official connector
 - ARM 10 or 20 pin, ST 14 pin, OCDS 16 pin
- JTAG has a very simple state machine
 - true power comes with a debug interface in between



<https://blog.senr.io/blog/jtag-explained>

- TCK
 - clock
- TMS
 - mode select via voltage
- TDI / TDO
 - data in and out
- TRST
 - optional to reset to “good” state



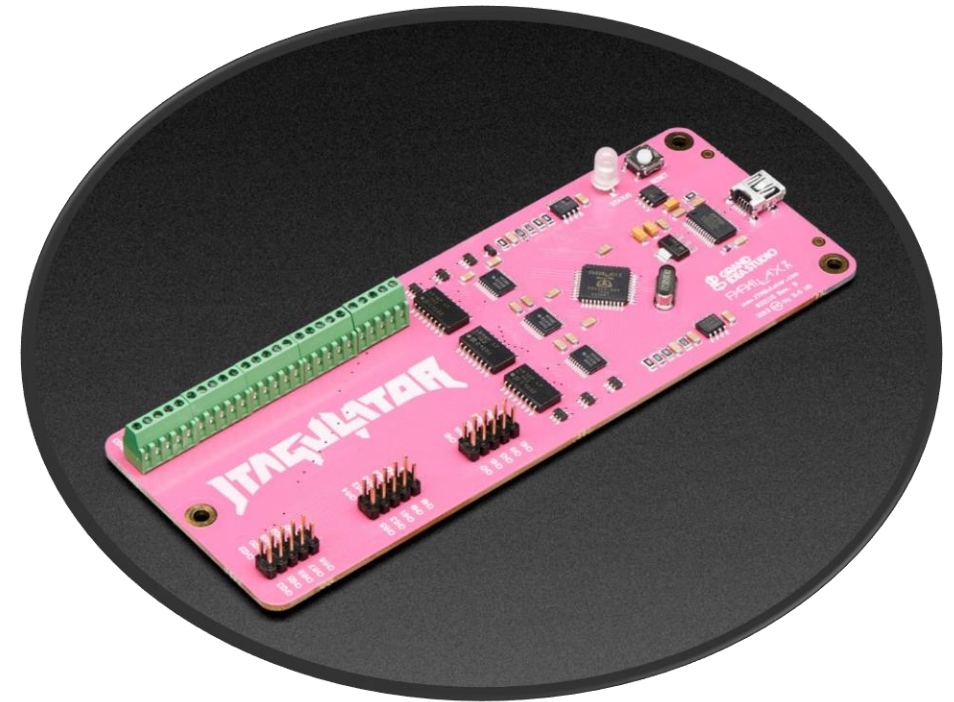
<https://blog.senr.io/blog/jtag-explained>

- Theoretically: Physical access to JTAG state machine is all you need
 - → easy: 4 pins, simple basic commands to influence state machine
 - but takes a lot of time to implement debug functionality
 - requires knowledge of internal chip design
- => Get access via software debugger, like OpenOCD
 - open source knowledge base for JTAG interfaces
 - huge collection of chips and interfaces
- What does OpenOCD ideally unlock?
 - code execution
 - memory access
 - breakpoint access
 - complete debugging access via gdb



EVALUATION OF DEBUG INTERFACES

- Hardware tool to brute force UART and JTAG pin layout
 - VDD/GND is required first
→ quite easy to find
 - 24 channels to analyse
 - open source hardware
- Check if we have access to JTAG FSM
- Usually no further tests required → open debug port is critical



<https://www.adafruit.com/product/1550>

NON-VOLATILE MEMORY INTERFACES



Serial EEPROM

- Non-volatile memory holds:
 - Bootloader code
 - Firmware (encrypted, signed, roll-back protection?)
 - Configuration
 - Root-of-Trust? (keys, e.g. for Boot Loader)

■ Attacks

- Firmware manipulation, downgrading attacks
- Reading static secrets
- Changing configuration



Memory programmer

■ Better solutions

- Different packages (BGA)
- Use SoC design → less accessible
- Dedicated secure memory/OTP for RoT



NAND Flash
(parallel interface)



VOLATILE MEMORY INTERFACES



DRAM in
SOJ-40 package



DRAM in
BGA package



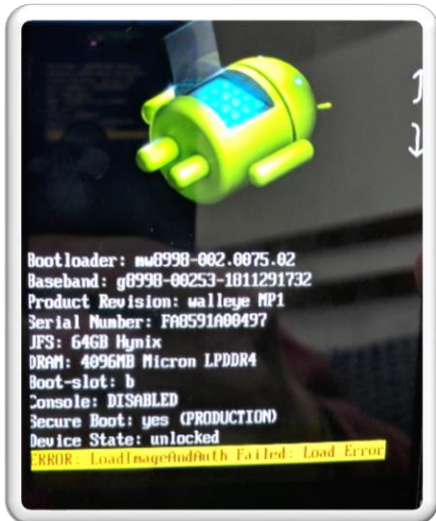
- Volatile memory (RAM) holds:
 - Program code and data during execution
 - Possibly security critical information (keys)

- Attacks
 - Probing... harder than with Flash/EEPROM (more pins, higher frequencies)
 - gets harder if a less-accessible package is used (e.g. BGA)
 - Cold boot attacks: if cooled, data persists > 1 hour in memory
→ read out after power was plugged

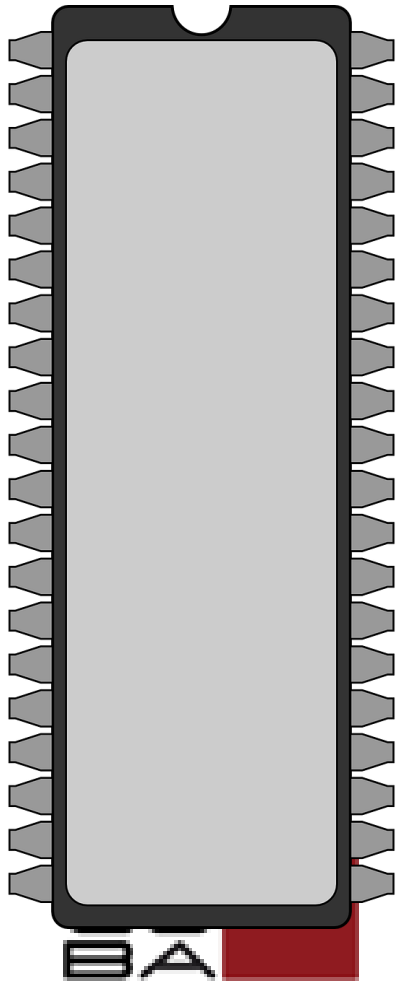
- Mitigations
 - SoC design makes probing harder
 - Memory encryption

FIRMWARE AND CONFIGURATION CORRUPTION

- Read/write firmware possible?
- Check if secure boot is enabled/used?
 - Does the device detect changes to the firmware/configuration?
 - Can we load an old firmware image?
- How does it react?



- Active field of research
- Static/dynamic code analysis requires source code (which is usually not available for IoT devices)
- → most of the time, it ends in reverse engineering
- Gathered information from firmware/source code image analysis
 - Which tools in use, programs running → known vulnerabilities
 - high level languages in use
 - library versions





■ binwalk

- find filetypes in image blobs
 - also recognizes compression
 - images, text files, file systems, etc.
- entropy analysis (encrypted/compressed image?)

■ radare2

- reverse engineering framework
- disassembler with large architecture support
- includes debugger support and scripting via python etc

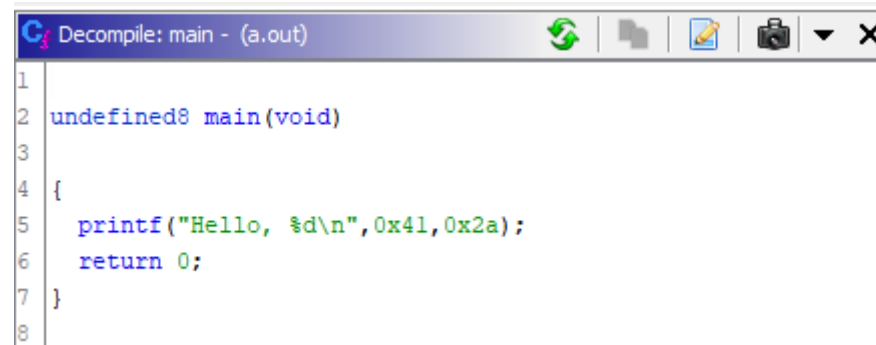
■ Ghidra

- free and open source reverse engineering tool by NSA
- alternative to IDA Pro
- supports partly ASM to C/C++ compilation



```
#include <stdio.h>

int main(void)
{
    int a = 42;
    int b = 23;
    printf("Hello, %d\n", a+b);
    return 0;
}
```



```
Decompile: main - (a.out)
1
2 undefined8 main(void)
3
4 {
5     printf("Hello, %d\n",0x41,0x2a);
6     return 0;
7 }
8
```

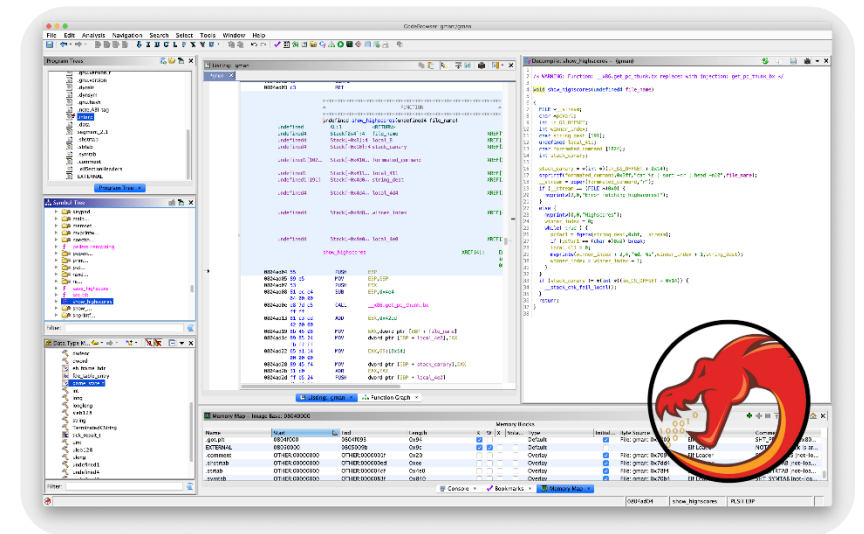
```
00101149 f3 0f le fa ENDBR64
0010114d 55 PUSH RBP
0010114e 48 89 e5 MOV RBP,RSP
00101151 48 83 ec 10 SUB RSP,0x10
00101155 c7 45 f8 MOV dword ptr [RBP + local_10],0x2a
                2a 00 00 00
0010115c c7 45 fc MOV dword ptr [RBP + local_c],0x17
                17 00 00 00
00101163 8b 55 f8 MOV EDX,dword ptr [RBP + local_10]
00101166 8b 45 fc MOV EAX,dword ptr [RBP + local_c]
00101169 01 d0 ADD EAX,EDX
0010116b 89 c6 MOV ESI,EAX
0010116d 48 8d 3d LEA RDI,[s_Hello,_%d_00102004]
                90 0e 00 00
00101174 b8 00 00 MOV EAX,0x0
                00 00
00101179 e8 d2 fe CALL printf
                ff ff
0010117e b8 00 00 MOV EAX,0x0
                00 00
00101183 c9 LEAVE
00101184 c3 RET
```



- Analyse/reverse engineer firmware → of course

But also...

- Find static credentials
 - Hardcoded username/password
 - Private keys
 - URLs/IPs
- Identify exploitable functions
 - E.g. memcpy-vehicle in TEE



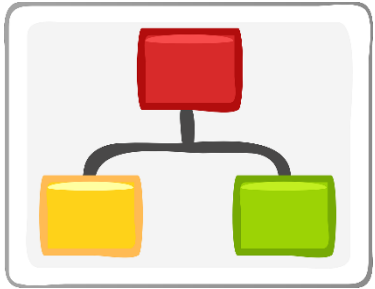
Manipulate the code, e.g. redirect to own server

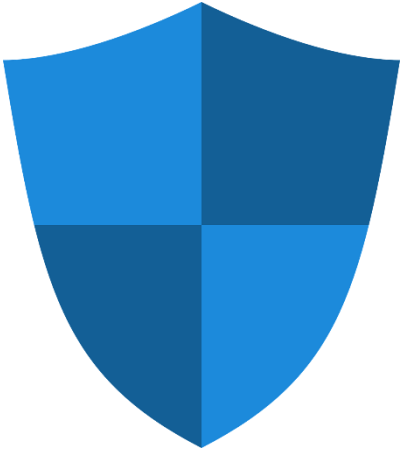
- Different markets addressed
 - low power (months or years on a button cell)
 - small size, low cost
 - compatibility with phones, tablets, computers
- 40 channels in BL frequency range (2.4 GHz)
 - 3 advertising channels
 - channel hopping simple to follow
 - allows sniffing
- Client / server architecture
 - IoT device is the server hosting services
 - Phone acts as a client requesting services
 - only one client per server at a time



BLUETOOTH LOW ENERGY CONNECTION FLOW

1. Server (IoT device) advertises regularly on advertisement channels
 → device name, service UUIDs, ...
2. Client sends CONNECT_REQ
 → frequency hopping sequence, connection interval, slave latency, supervision timeout, ...
 → after this packet, the connection is established, server is seen as slave, client as master
3. Client can use host layer via GATT (Generic Attribute Profile)
 → write, read, notify, subscribe, ...





- BLE can be encrypted via AES-128
 - Cipher Block Chaining-Message Authentication Code (CCM) Mode

- Devices must first pair
 - how is the key exchanged?
 - discovery and connection process are always unencrypted

- In practice?
 - hardly any IoT device gets this right (hardcoded keys, ...)
 - or doesn't use encryption at all

- Sniff and fuzz to evaluate





- Hardware for BLE testing is cheap
 - <https://www.nordicsemi.com/Software-and-Tools/Development-Kits/nRF52840-DK>
 - <https://www.nordicsemi.com/Software-and-tools/Development-Kits/nRF52840-Dongle>
- Enumeration can be done via mobile phone or with above HW
 - <https://play.google.com/store/apps/details?id=no.nordicsemi.android.mcp>
 - <https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Connect-for-desktop>
- Sniffing tool available as Wireshark plugin
 - <https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Sniffer-for-Bluetooth-LE>
- MITM tools available online as free software
 - <https://github.com/securing/gattacker>



Secure Product Lifecycle

Penetration Testing for IoT Devices –
A Hardware Evaluator’s Perspective

WHEN YOU NEED TO BE SURE

SGS