Reverse-Engineering the Supra iBox

Exploitation of a hardened MSP430-based device

black hat
USA 2014

ACCUVANT

Who am I

Braden Thomas

- Senior Research Scientist, Accuvant
- Primarily focus: embedded devices, reverse-engineering, exploit development
- Previously worked at Apple Product Security
  - *Software* background

Agenda

- What is the iBox?
- Android app
- Opening the device
- Firmware extraction: techniques used and tried
- Findings
- Demo

Why is this interesting?

- Devices attempting to to store crypto secrets in general-purpose microcontrollers
- Just because it's cheap and easy, it's not necessarily smart
  - iBox is a case study of why
- Hack into houses…
  - Over Bluetooth!

Supra iBox

- Real estate physical key container
- #1 in market, main competition is SentriLock

iBox

iBox BT

iBox BT LE

Keys

- ActiveKEY
- Cell radio

- eKey: iOS/Android app
- Dongle/Keyfob for Bluetooth/IR

Android App

eKey Android app

- Focused on authentication algorithm
- Each eKey has a serial number and a “syscode”
  - Syscode is an integer corresponding to regional market (e.g. Atlanta)
- Serial number/Syscode are required at first app launch in an obfuscated blob

eKey Android app

- Serial number/syscode are used as credential to speak to back-end web service
- Web service provides authentication “cookies" (binary blobs of data)
- App transmits cookies to the iBox over Bluetooth/IR
- Must provide PIN code (associated with serial number/ syscode) to open the lock

Programmed auth flow

- Two authentication modes:
  - *Programmed* and *deprogrammed* authentication
- Programmed authentication used exclusively in the field
  - Send IDENTITY cookie
  - Send CONFIGURATION cookie
  - Send OBTAIN KEY message
  - Send KEYAUTH cookie
  - Send DEVICE OPEN message

Programmed auth

- All cookies contain AES MACs so cannot be modified

- eKey also sends “update bytes” which change daily
  - Only available from Supra server (AES MAC)

- eKey can generally only open iBox in same syscode

Must access firmware

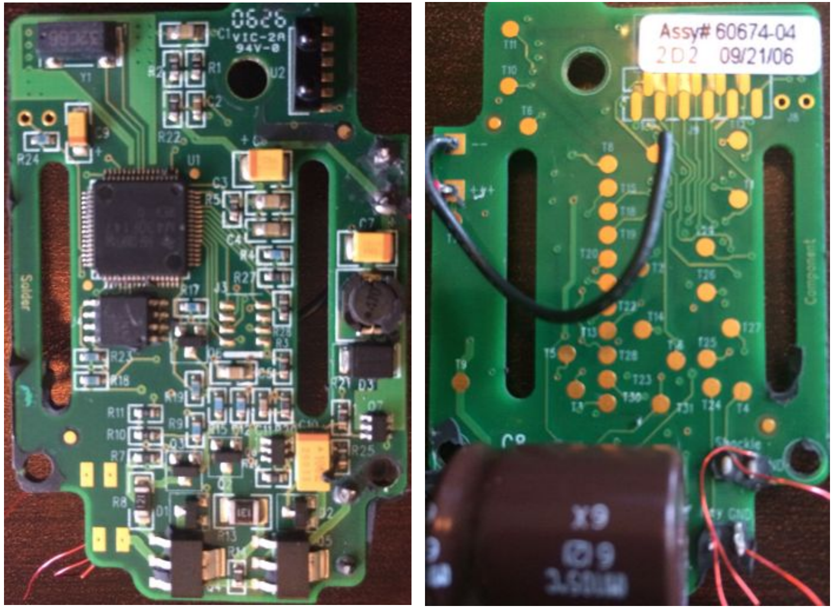
- Attacker doesn’t have a valid serial/syscode
- Even if obtained one (social engineering), don’t have keyholder’s PIN
- And doesn’t want to communicate with Supra’s server to obtain cookies

Opening the Device

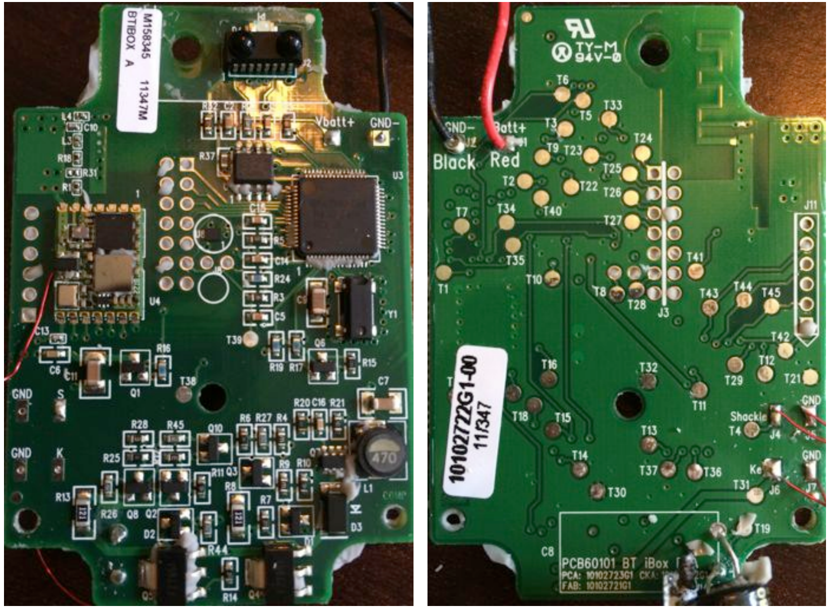
Physical access

- iBox:
  - – Cut off hard plastic shell
  - – Remove hex screws
  - – Open key container
    - Use legitimate eKey or exploit
- iBox BT: (above, plus)
  - – Cut off shackle
  - – Must pop rivets (big pain!)

Board photos



iBox



iBox BT

Internals

iBox:

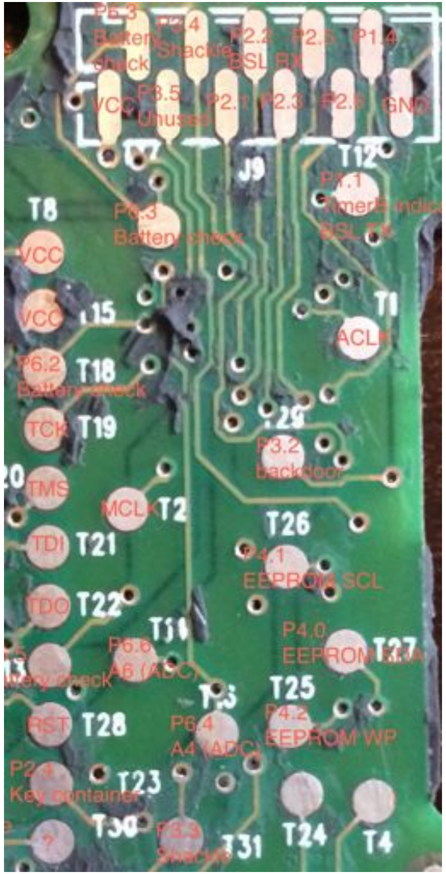
- MSP430F147
- TFBS4710 serial IR transceiver
- 24LC256 serial EEPROM

iBox BT:

- MSP430F248
- STMicroelectronics bluetooth serial module
- Atmel EEPROM

Reverse-engineering steps

- Focus on iBox
  - Board easier to obtain (no annoying rivets)
  - Older software more likely to be insecure
  - Keys are the same anyway!
- Map-out the test pads
- Find debugging interfaces
- Perform firmware extraction



Firmware Extraction

MSP430 firmware extraction

- **JTAG**
  - 4-wire and 2-wire
  - MSP430F147 only supports 4-wire
  - JTAG security fuse is blown, prohibiting JTAG

- **BSL**

BSL Overview

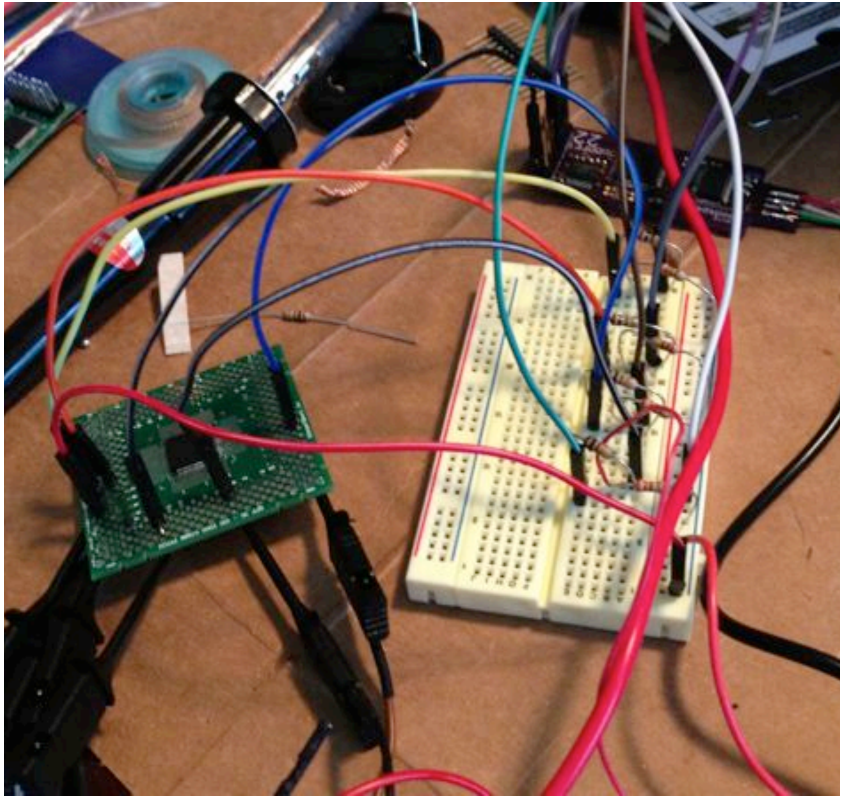
- “Bootstrap loader”
- Serial interface
- Permits read/write access to flash memory
- Implemented with code stored in special flash region
- Nearly all acccess is restricted with password
  - Interrupt vector table is used: inherently unique and secret
  - Only mass-erase can be performed without password

Existing BSL attacks

- Travis Goodspeed: “Practical Attacks Against the MSP430 BSL” in 2008
  - Voltage glitching attack
  - BSL password comparison timing attack

Voltage glitching attack

- Used GoodFET22 with ADG1634 + DAC for glitching during authentication check
- Remove the chip from the board to avoid interference
- Step down power on all lines using resistors
- Only feasible on BSL 1.x to avoid mass-erase on incorrect password
  - MSP430F147 has BSL 1.1



Results of voltage glitching

- Failed to reproduce
- Device continued running undeterred or died altogether
- GoodFET's MSP430 is too slow to glitch another MSP430
  - BSL runs at 1Mhz, and GoodFET (MSP430F2618) can be clocked up to 16Mhz

BSL timing attack

- Password byte comparison has a single clock-cycle timing difference between the "correct" and "incorrect" paths
- Send each byte ([0x00-0xff] x 32) and measure # of clock cycles to determine byte makeup of password

```
ROM:0CDA handle_tx_passwd:                               ; CODE XREF: sub_E10-1B8'j
ROM:0CDA                 mov.w   #0FFE0h, R6             ; IVT address (correct password)
ROM:0CDE                 mov.w   #20h, R7                ; pw len
ROM:0CE2
ROM:0CE2 check_next_byte:                                ; CODE XREF: sub_E10-11A j
ROM:0CE2                 call    #rx_byte
ROM:0CE6                 cmp.b   &received_byte, 0(R6) ; compare byte by byte
ROM:0CEC                 jz      equal_byte
ROM:0CEE                 bis.b   #WILL_SEND_NAK, &bsl_state ; bad pw bit
ROM:0CF2
ROM:0CF2 equal_byte:                                     ; CODE XREF: sub_E10-124'j
ROM:0CF2                 inc.w   R6
ROM:0CF4                 dec.w   R7
ROM:0CF6                 jnz     check_next_byte
```

BSL 1.10

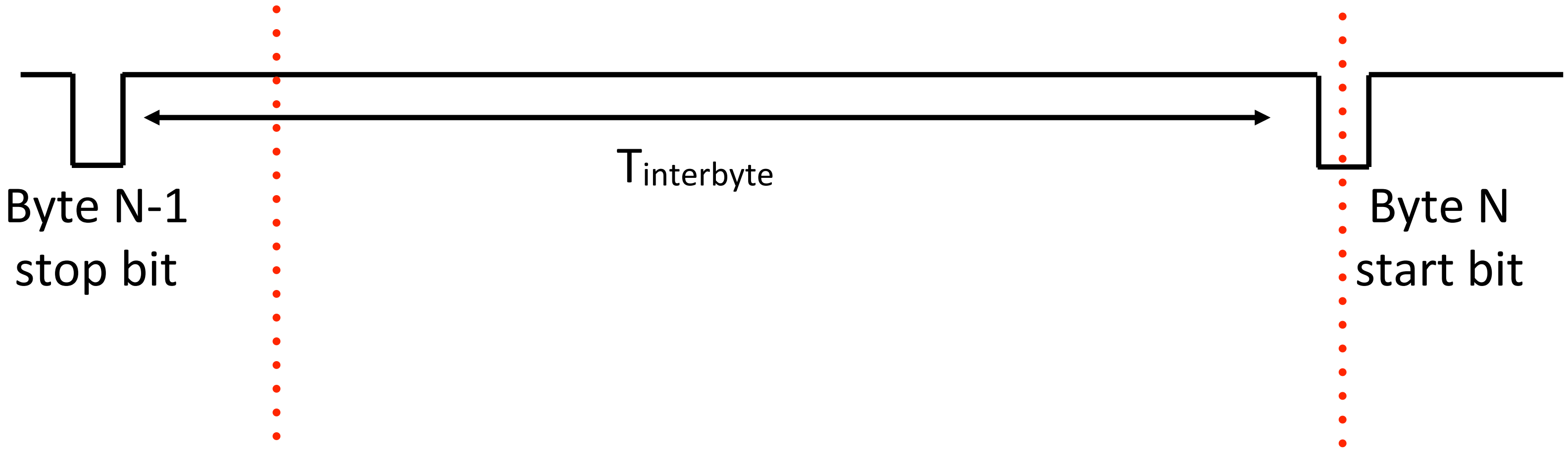
Timing attack problems

- 1 start bit, 8 data bits, parity bit, 1 stop bit
- Bit-banged
- Between bytes, will *wait* for start bit to go low when receiving

```
ROM:0F2E bitcnt_is_0:                              ; CODE XREF: ROM:0F32 j
ROM:0F2E                 bit.b   #BIT2, &P2IN
ROM:0F32                 jnz     bitcnt_is_0
```

- If this loop executes > 1 time, you have destroyed all prior timing information
- Device will *check* that RX line after stop bit is high, or cause an error

Timing attack problems



Byte N-1
stop bit

$T_{interbyte}$

Byte N
start bit

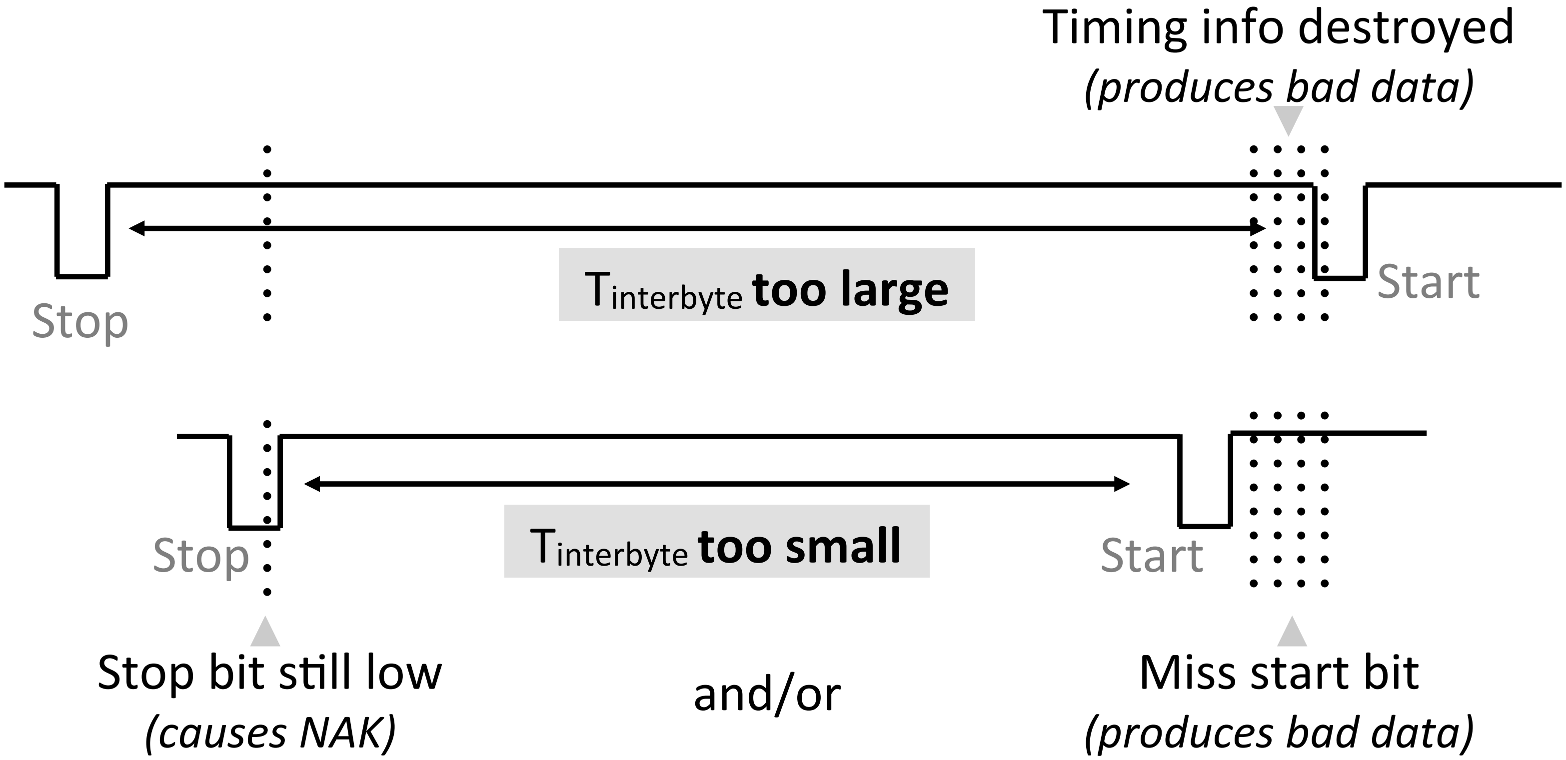
Device checks
here

Timing info
destroyed here if not low

Timing attack problems

- Ideal $T_{interbyte}$ = number of instructions * clock speed
  - Clock speed is highly inconsistent
    - BSL uses DCOCLK (software clock), cannot force crystal
  - Number of instructions varies
    - Due to timing vulnerability
- Any mistakes are multiplied 34x (since 34 post-header bytes per auth)

Timing attack problems



Timing info destroyed
(produces bad data)

Stop

$T_{interbyte}$ **too large**

Start

Stop

$T_{interbyte}$ **too small**

Start

Stop bit still low
(causes NAK)

and/or

Miss start bit
(produces bad data)

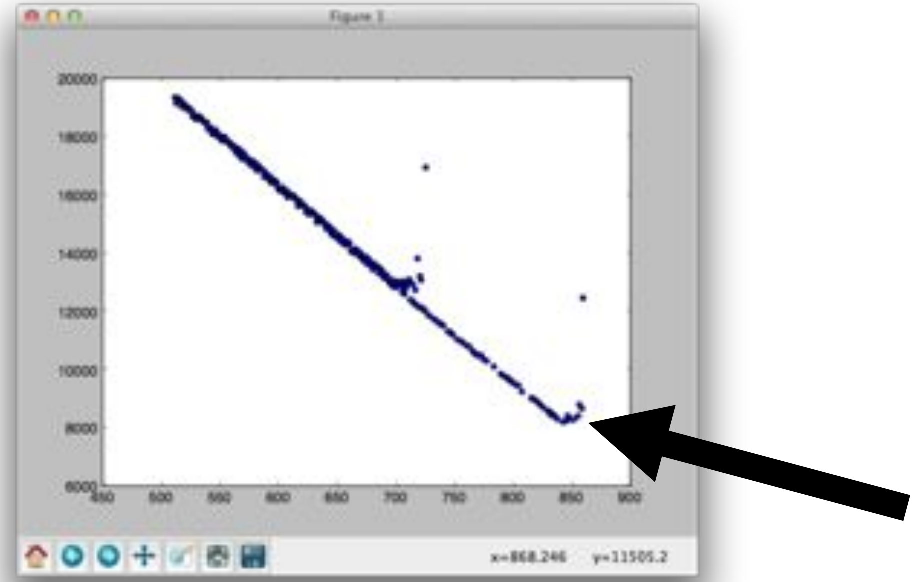
Timing attack problems

- If timing is bad, you will receive a NAK response
- Since password is inherently wrong, you will receive a NAK response
- No good way to differentiate between the NAKs!

Timing attack game plan

- Test with same-model chip (with known BSL password) to find ideal timing
- Use external crystal on GoodFET to eliminate attacker-side clock problems
- Slowly decrease $T_{interbyte}$ until correct password is no longer ACKed
  - Find the run with the lowest overall total time
  - You have found ideal $T_{interbyte}$
  - Re-use on target chip

Timing attack results



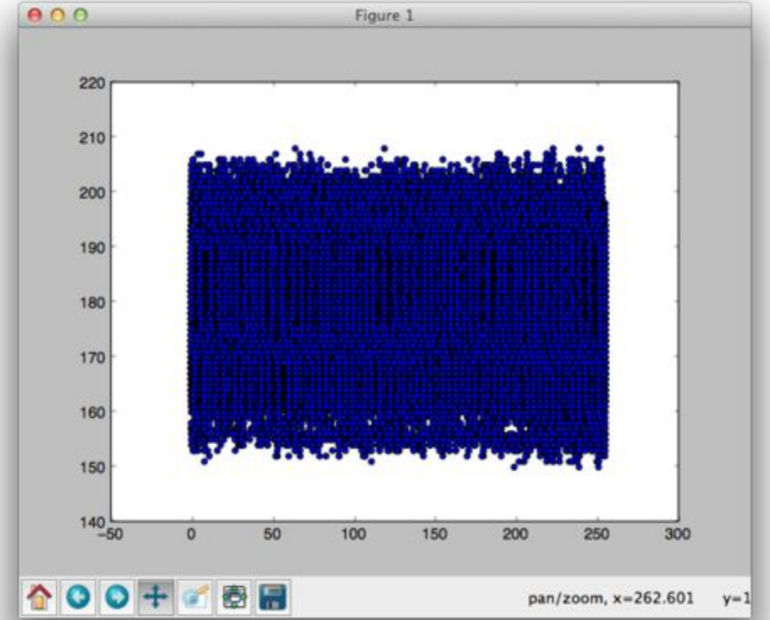
ideal $T_{interbyte}$

Total time vs decrease in $T_{interbyte}$

Timing attack results

- Looks good at macro level
- Wildly inconsistent at micro level
- Overall total times will vary by thousands of attacker clock cycles
- Tried modifying BSL to expose bit read time on a line
- Tried just focusing on last byte: only need to get three $T_{\text{interbyte}}$ correct
  - last byte + checksum

Modified attack results



Guessed byte vs overall time

Timing attack conclusions

- Attack was a failure
- Likely due to DCOCLK inconsistencies during the *tare* routine, which produces victim chip's timing for serial communication (length of "sleep"s)
- If this tare routine value is inconsistent, the timing used for *every serial bit* will differ, multiplying errors
- Doesn't appear to average out in the short term

“Paparazzi” attack

- Firmware extraction technique
  - Goodspeed told me about this
  - Permits bypassing JTAG security fuse
  - Most likely due to photoelectric effect

MSP430 JTAG security

- MSP430F1xx/2xx/4xx: physical fuse
  - Once blown ("programmed"), it's blown
- MSP430F5xx/6xx: electronic fuse mechanism
  - Can be unprogrammed by erasing 0x17fc
  - Not successful at attacking these

MSP430 1/2/4xx fuse

- Fuse check is performed by toggling TMS line twice with TDI high
- Current is measured from TDI across the fuse

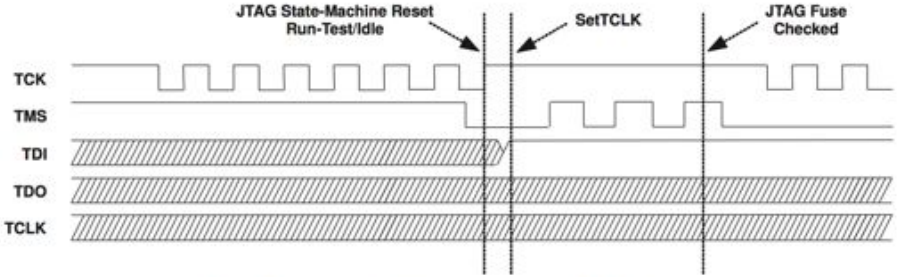
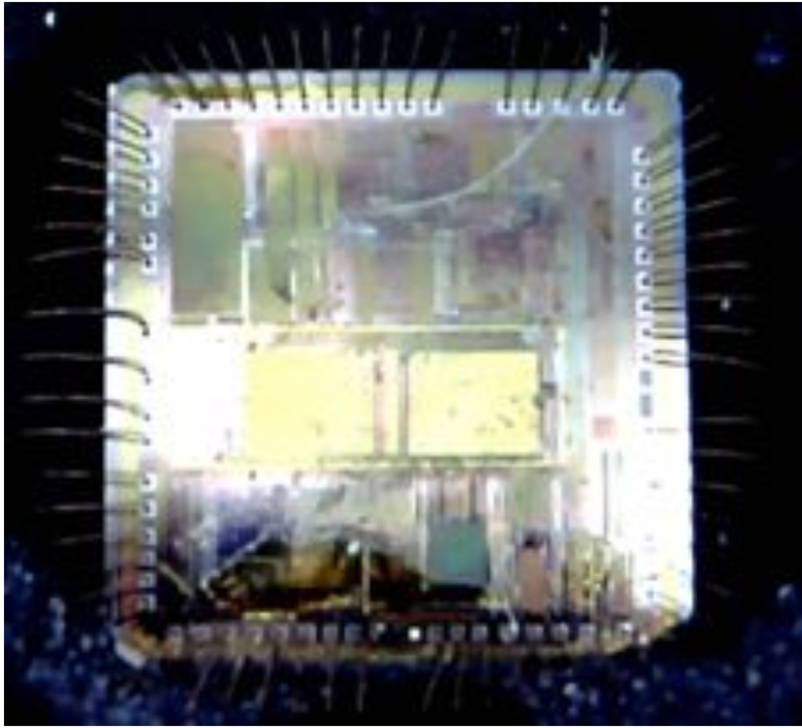


Figure 1-12. Fuse Check and TAP Controller Reset

Chip logic remembers the result

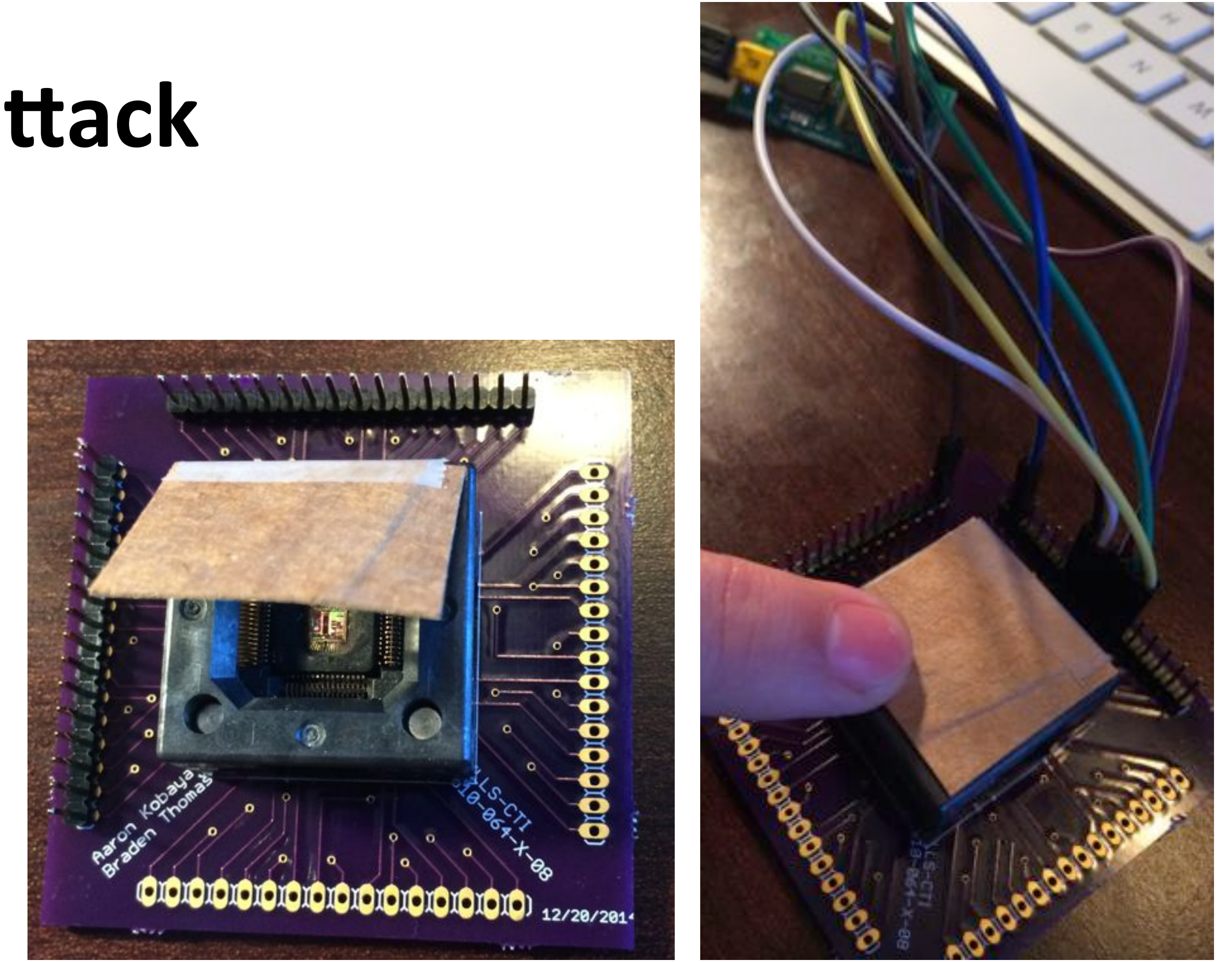
“Paparazzi” attack



- Decap the chip
  - Ensure bonding wires remain intact
    - Jet etching may be required
  - <$100 outsourced to lab
- Run a tight JTAG loop on reset-tap + fuse-check
- Every ~200 iterations attempt authenticated action
  - Read first address in BSL memory space

“Paparazzi” attack

Expose the die and hit with camera flash



“Paparazzi” attack

- When valid data returned, success!
- Do not power the chip down, or flip reset line
  - Requires GoodFET software modification
- Be sure to power the chip externally during attack
- Don’t expect chip to be in normal state
  - I usually just read BSL password then reset

“Paparazzi” attack: Why?

- JTAG fuse check works by measuring current across fuse
  - Photoelectric effect causes transistor to release electrons when struck with photons
  - Causes current to appear to pass across the fuse
  - Alternative theory is UV erasing memory cell where JTAG state stored (e.g. bunnie’s attack on PIC microcontroller), but digital camera flash produces minimal UV and attack is instant

Paparazzi Demo

FINDINGS

MSP430 firmware reversing

- Calling convention
  - R12
  - R14
  - Remaining arguments pushed to stack
  - Return: R12
    - Occasionally R13 is also used, if 32-bit return

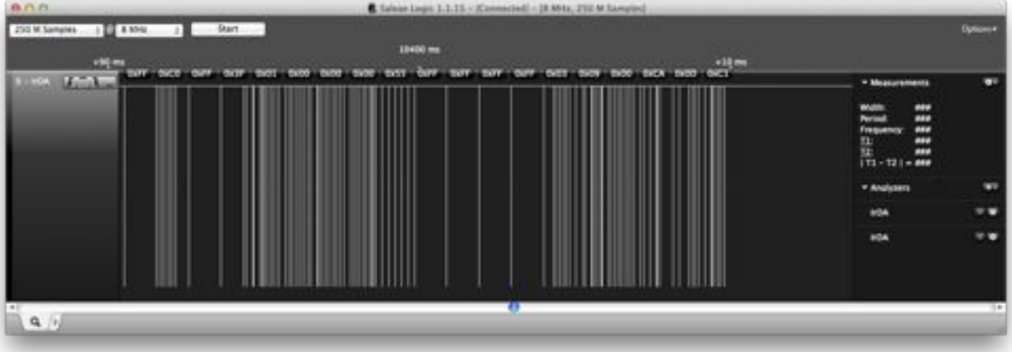
MSP430 firmware reversing

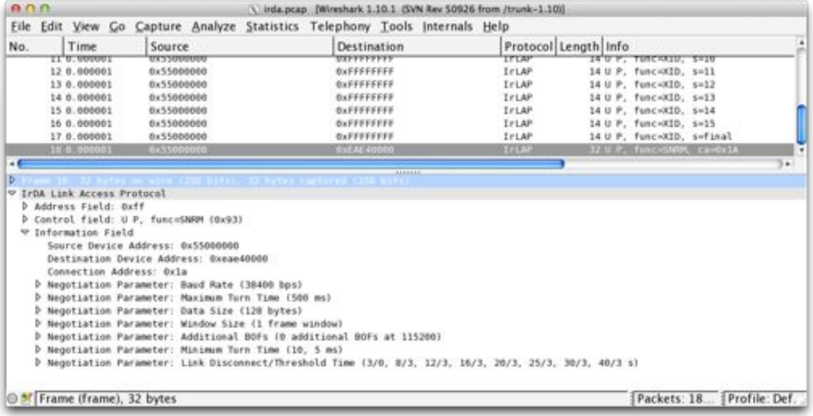
- Only unique thing was “sparse index” switch statement construction
- Used a common helper function that reads function return address off the stack, then parses data structure after call to find out jump destination

```
seg001:0000A7F4                 mov.b   &command_id_byte, R12
seg001:0000A7F8                 call    #switch_statement_helper
seg001:0000A7F8 ; ---------------------------------------------------------------
seg001:0000A7FC                 .short loc_AA44         ; default
seg001:0000A7FE                 .short 25
seg001:0000A800                 .byte 48
seg001:0000A801                 .short handle_connection_start
seg001:0000A803                 .byte 49
seg001:0000A804                 .short handle_send_identity
seg001:0000A806                 .byte 50
seg001:0000A807                 .short handle_send_configuration
seg001:0000A809                 .byte 51
seg001:0000A80A                 .short handle_crypto_key_update
seg001:0000A80C                 .byte 52
seg001:0000A80D                 .short handle_base_challenge_response
seg001:0000A80F                 .byte 53
```

IrDA

- Surprisingly large (~25%) amount of firmware dedicated to IrDA
- Bit-banged serial-ish with short pulse width
- Can be sniffed from test pad on board and decoded with custom Logic plugin
- Export from Logic, post-process with python into pcap, and Wireshark does the rest





```
No.  Time       Source        Destination   Protocol Length Info
  12 0.000001   0x55000000    0xFFFFFFFF    IrLAP      14 U P, func=XID, s=11
  13 0.000001   0x55000000    0xFFFFFFFF    IrLAP      14 U P, func=XID, s=12
  14 0.000001   0x55000000    0xFFFFFFFF    IrLAP      14 U P, func=XID, s=13
  15 0.000001   0x55000000    0xFFFFFFFF    IrLAP      14 U P, func=XID, s=14
  16 0.000001   0x55000000    0xFFFFFFFF    IrLAP      14 U P, func=XID, s=15
  17 0.000001   0x55000000    0xFFFFFFFF    IrLAP      14 U P, func=XID, s=final
  18 0.000001   0x55000000    0xEAE40000    IrLAP      32 U P, func=SNRM, ca=0x1A

▷ Frame 18: 32 bytes on wire (256 bits), 32 bytes captured (256 bits)
▽ IrDA Link Access Protocol
  ▷ Address Field: 0xff
  ▷ Control field: U P, func=SNRM (0x93)
  ▽ Information Field
      Source Device Address: 0x55000000
      Destination Device Address: 0xeae40000
      Connection Address: 0x1a
    ▷ Negotiation Parameter: Baud Rate (38400 bps)
    ▷ Negotiation Parameter: Maximum Turn Time (500 ms)
    ▷ Negotiation Parameter: Data Size (128 bytes)
    ▷ Negotiation Parameter: Window Size (1 frame window)
    ▷ Negotiation Parameter: Additional BOFs (0 additional BOFs at 115200)
    ▷ Negotiation Parameter: Minimum Turn Time (10, 5 ms)
    ▷ Negotiation Parameter: Link Disconnect/Threshold Time (3/0, 8/3, 12/3, 16/3, 20/3, 25/3, 30/3, 40/3 s)

Frame (frame), 32 bytes     Packets: 18...     Profile: Def.
```

Firmware reversing finds

1. How Supra crypto *really* works
2. Actually *three* authentication modes
3. Hardware backdoor!
4. Memory read/write command permits reading/writing flash using hidden mode

①

Supra crypto architecture

- All crypto keys used are derived from or encrypted with two keys (AES128)

- **Device Key**
  - Rarely used in the field, used to get high authentication level (i.e. for "deprogramming" a device to use it in another syscode region)

- **Syscode Key**
  - Root of trust for all normal operations (e.g. opening the key container)
  - Shared by entire geographical region

- *Neither are ever accessible to the eKey app or readable via remote commands*

②

Syscode Key

- Provisioned during unknown process at local MLS office
  - Device must be in *deprogrammed* mode
  - They must have some authenticated channel to obtain the syscode key for their region

- A MAC key and an Encryption key are derived from syscode key, and used to validate cookie integrity and decrypt other ephemeral keys

- Compromising this key permits attacker to generate fake "authentication cookies"
  - Can open any lock in geographical region without leaving a trace

②

Third authentication mode

- Permits access to visitor log in EEPROM
  - Useful if the lock has been unlocked before
- Requires **<u>no</u>** authentication cookies for access
- Visitor log contains the serial number/syscode of connecting eKeys
  - This solves one of our earlier problems, but still need PIN to use

③

Brute Force

- PIN only 4 digits
- However device has PIN brute-force protection
  - eKey will get "locked out" and cannot communicate for 10m
  - Exhaustive PIN brute force would take about 1 week waiting for lockouts
  - However, lockout counter stored in EEPROM and can be erased with physical access

③

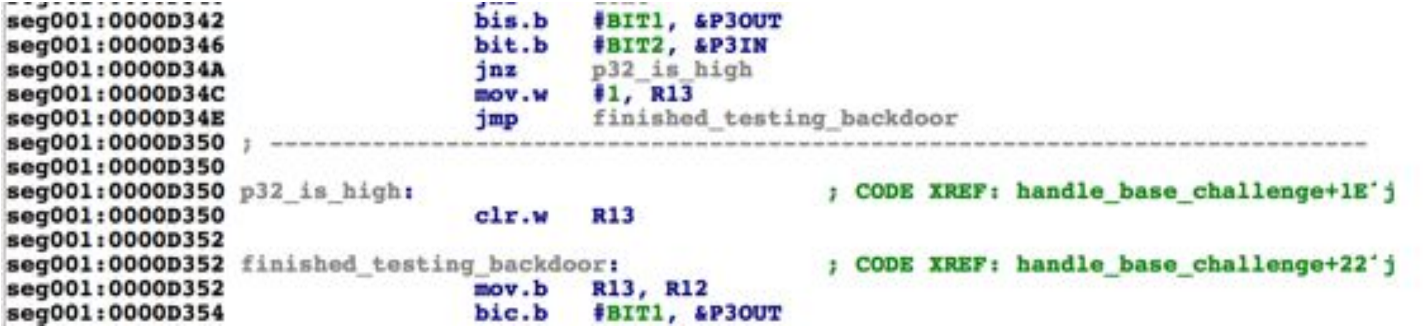
Hardware backdoor

- *Deprogrammed* authentication
  - Android app only uses this method when device is deprogrammed
- Can actually be used when device is programmed if you know the Device Key
  - Highest access mode, permits overwriting keys
  - Likely used by MLS office, they must have a secure channel to get Device Keys for their devices
- Implementation contains hardware backdoor

③

Hardware backdoor

- P3.1 goes high
- Immediately test P3.2
- If low, backdoor is in effect

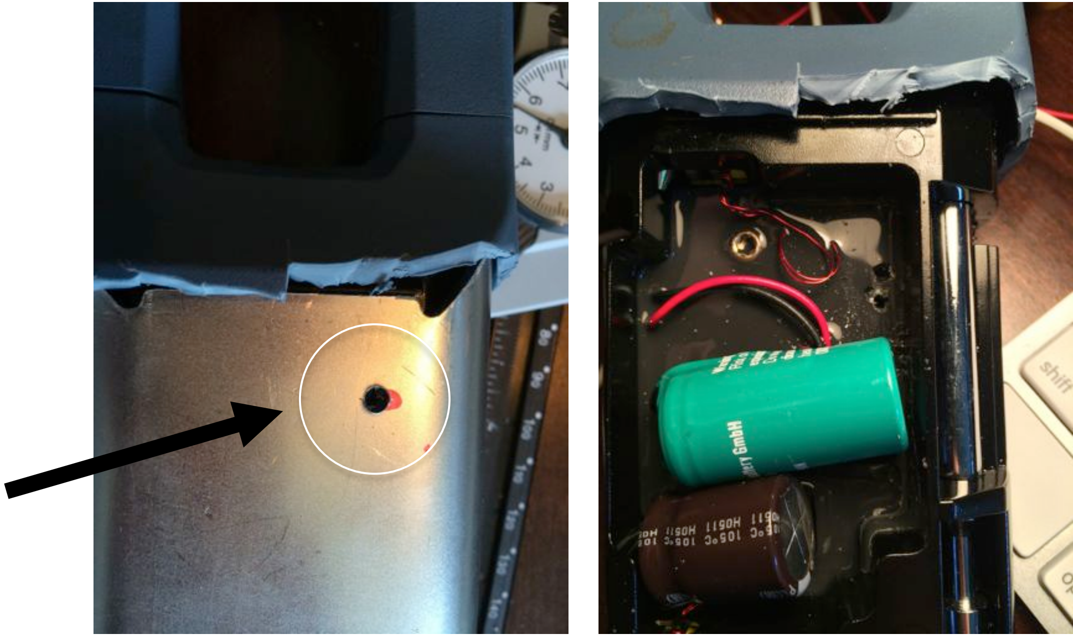


```
seg001:0000D342                 bis.b   #BIT1, &P3OUT
seg001:0000D346                 bit.b   #BIT2, &P3IN
seg001:0000D34A                 jnz     p32_is_high
seg001:0000D34C                 mov.w   #1, R13
seg001:0000D34E                 jmp     finished_testing_backdoor
seg001:0000D350 ; ---------------------------------------------------------------------------
seg001:0000D350
seg001:0000D350 p32_is_high:                            ; CODE XREF: handle_base_challenge+1E`j
seg001:0000D350                 clr.w   R13
seg001:0000D352
seg001:0000D352 finished_testing_backdoor:              ; CODE XREF: handle_base_challenge+22`j
seg001:0000D352                 mov.b   R13, R12
seg001:0000D354                 bic.b   #BIT1, &P3OUT
```

③

Hardware backdoor

- P3.1 and P3.2 are connected to each other (through a resistor)
- Desolder the resistor and you can bypass per-device authentication
- Destroy the resistor with a single drill hole in back of closed iBox and you can open it up with deprogrammed auth



④

Flash write+erase attack

- Way to extract Syscode Key without decapping?
- Keys are in "Information Memory" which is erased by BSL mass-erase
- Generally, must erase flash between writes
- iBox has Memory Write command that permits writing to same information memory segment where keys are stored
  - Entire segment is copied to stack buffer, Flash segment is erased, modified, and then written back
  - Stack is in RAM… which is *not* erased by BSL mass-erase

④

Flash write+erase attack

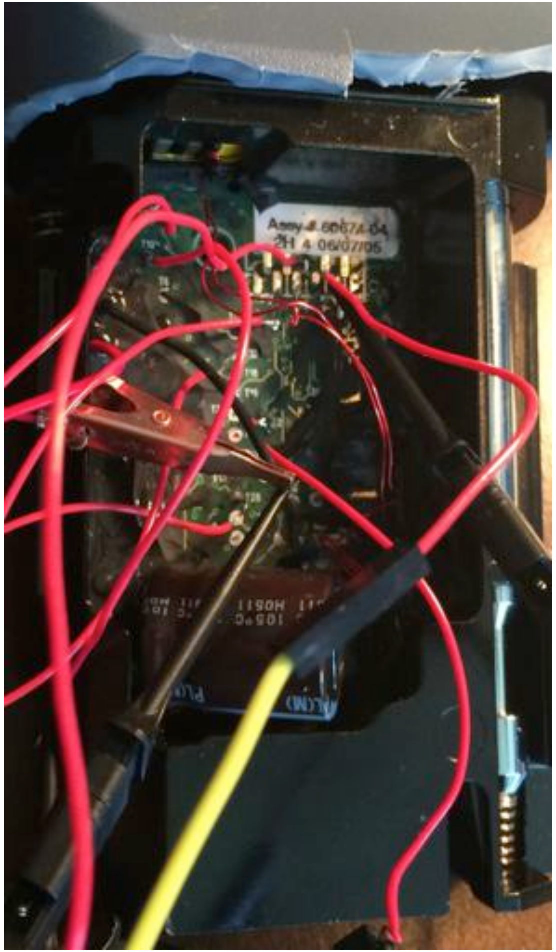
- First use hardware backdoor to “authenticate”
- Initiate a Memory Write command to information page (at an unused location)
- Information page will be copied to stack buffer, modified, and written back to flash
- Quickly reset device and perform mass-erase of flash via BSL
- Read RAM using BSL (using default password)

④

Flash write+erase attack

- **Great success!**

- Special GoodFet application that counts clock cycles
  - Run application right before sending iBox Memory Write command
  - Send Memory Write command
  - Application will reset chip and put into BSL mode
  - Subsequently can mass-erase and read RAM
  - Attack can only be performed once, but Syscode Key is obtained



Demo

Conclusions/solutions

- Supra
  - Discussed issues with them in June
  - Very receptive, started working on fixes
  - Starting to deploy solution in <60 days
- Other applications:
  - Avoid storing cryptographic secrets in general purpose microcontrollers flash memory

Greetz

- Hardware socket by Aaron Kobayashi
- Thanks to Nathan Keltner and Kevin Finisterre
- Thanks to Travis Goodspeed for prior work

Questions