Authenticating Micro-controllers

P. Schaumont

Bradley Department of Electrical and Computer Engineering
Virginia Tech
Blacksburg, VA

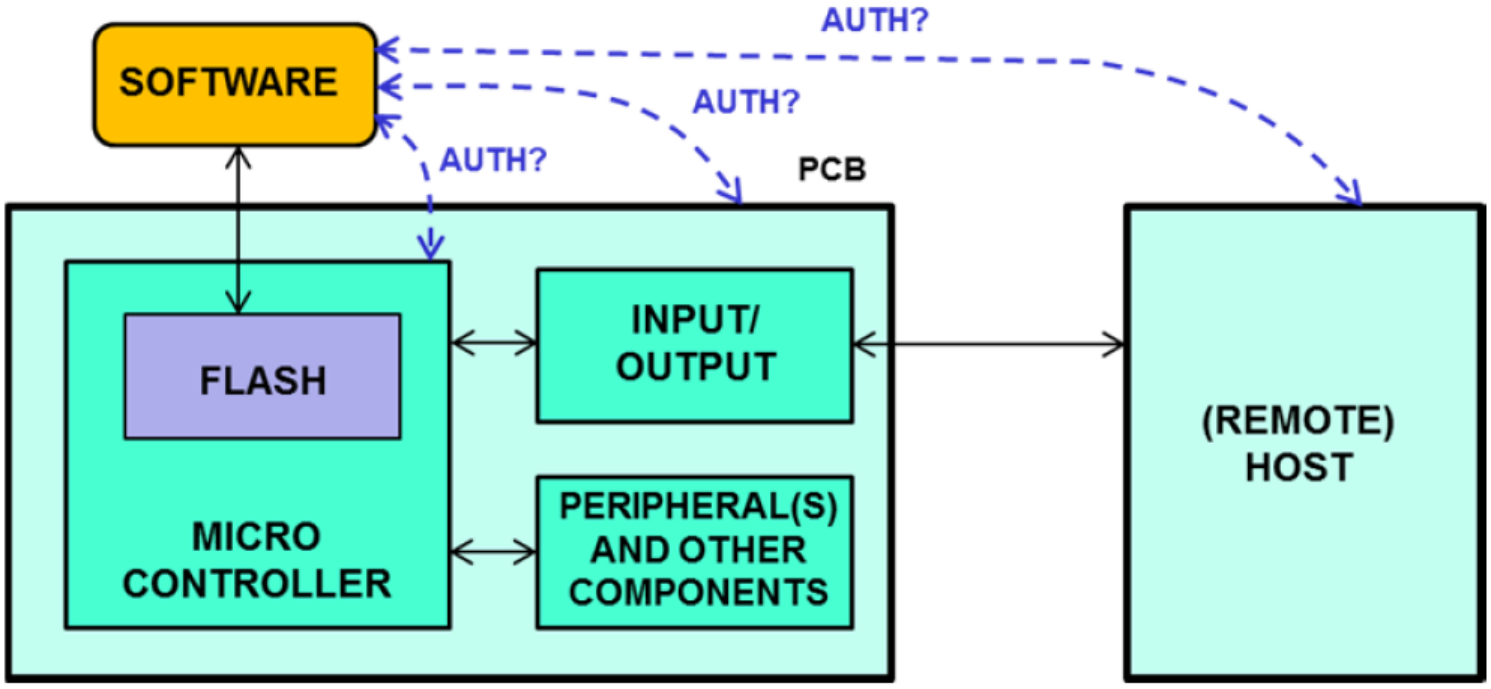
Design and Security of Cryptographic Functions, Algorithms and Devices, 2013

Objectives of this presentation

- How to support authenticity on microcontrollers?
  - Firmware support for authentication protocols
  - Signed firmware upgrades
- Coding examples, sample projects

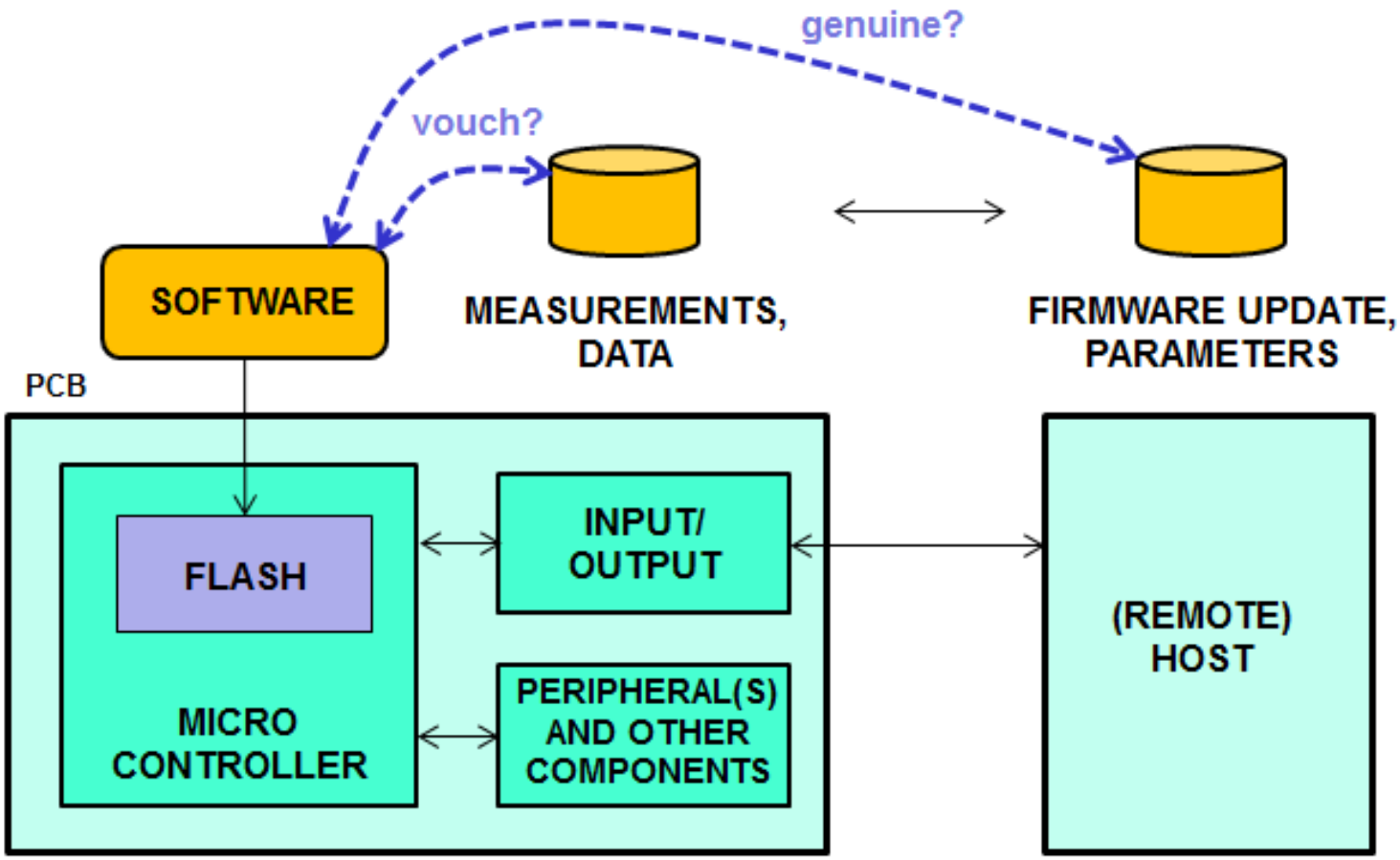
Embedded Authentication

(1) Ensure that server, environment, hardware is genuine



Embedded Authentication

(2) Ensure that data items, firmware downloads, are genuine



Microcontroller technologies

We develop authentication in the context of the following technologies

- Single-chip implementation with CPU, RAM, Flash, Peripherals
- Lightweight processing platform (8/16 bit)
- Dedicated toolchain for *bare-metal* C programming
- May or may not be always-on, which affects persistent state

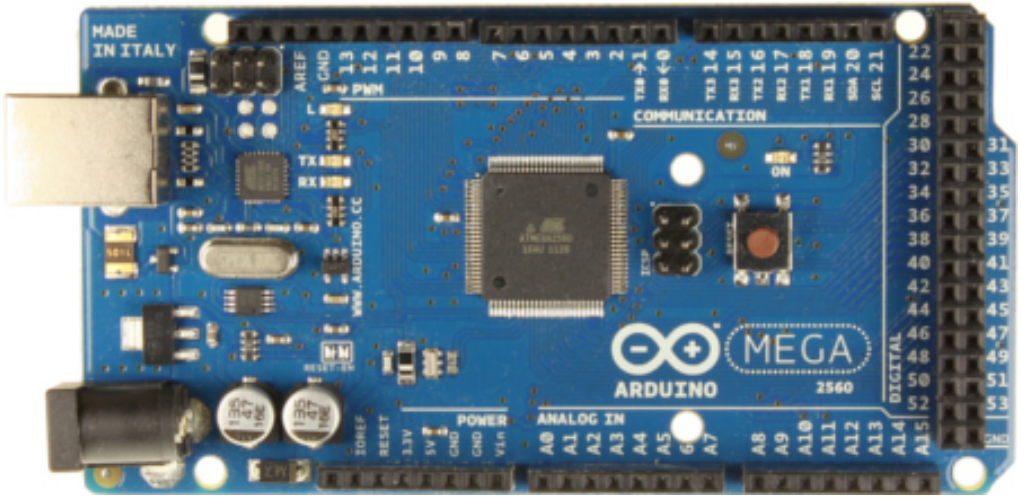
Security assumptions

- Chip package is the trust boundary
- Correctly-designed firmware prevents code injection
- No implementation attacks

Example: ATMega2560

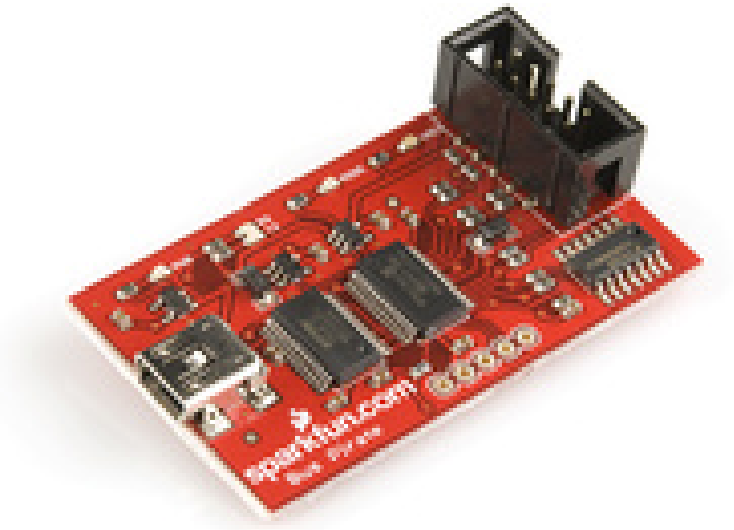
8-Bit Microcontroller

- AVR CPU
- 256KB Flash, 4KB EEPROM, 8KB RAM
- Lock bits restrict access to non-volatile memory
- Timers, PWM, ADC, SPI, UART, ...
- AVR LibC (gcc) toolchain `http://www.nongnu.org/avr-libc/`



Example: ATMega2560 (Support Hardware)

Bus Pirate (for I/O)



JTAG ICE (for firmware loading and debugging)

Example: CC430F6137

16-Bit Ultra-Low-Power MCU

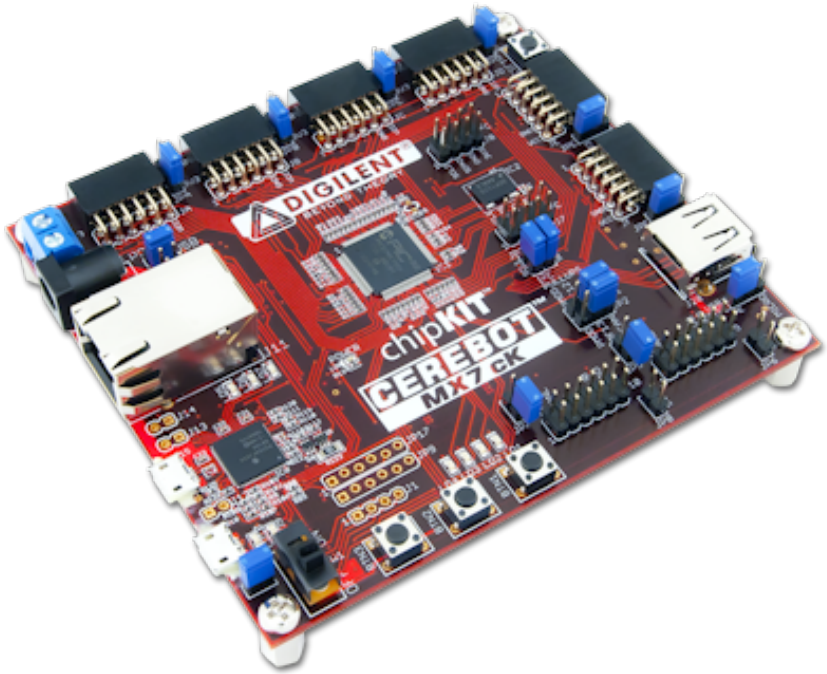
- MSP430 CPU
- 32KB Flash, 4KB RAM
- Timers, 12-bit A/D, T/V sensor, sub-1GHz RF
- 32-bit Hardware Multiplier, AES
- mspgcc toolchain

  `http://sourceforge.net/apps/mediawiki/mspgcc`

Example: PIC32MX795F512L

32-Bit Microcontroller

- MIPS CPU
- 512+12KB Flash, 64KB RAM
- Timers, USB, CAN, ADC, SPI, UART, ETH, I2C, ...
- MSPlabX toolchain `http://www.microchip.com/mplabx/`



Basic One-way Authentication

- Prover `P`, Challenger `C`, pre-shared secret key `K`
- `C` $\leftarrow$ `P`: Identifier `ID`
- `C` $\rightarrow$ `P`: Nonce `N`
- `C` $\leftarrow$ `P`: `encrypt(K, ID || N)`
- `C` verifies encryption of `(ID || N)`

Important Requirements

- Nonce must be unique, otherwise replay is possible
- Preshared key K is a system-wide secret (liability)

Basic One-way Authentication

- Prover `P`, Challenger `C`, pre-shared secret key `K`
- `C` $\leftarrow$ `P`: Identifier `ID`
- `C` $\rightarrow$ `P`: Nonce `N`
- `C` $\leftarrow$ `P`: `encrypt(K, ID || N)`
- `C` verifies encryption of `(ID || N)`

Important Requirements

- Nonce must be unique, otherwise replay is possible
- Preshared key K is a system-wide secret (liability)

Basic Mutual Authentication

- Prover/Challenger `P1/C1`, `P2/C2`, pre-shared secret key `K`
- `P1/C1` $\leftarrow$ `P2/C2`: Identifier `ID2`, Nonce `N2`
- `P1/C1` $\rightarrow$ `P2/C2`: Nonce `N1`, `encrypt(ID1 || N2)`
- `P1/C1` $\leftarrow$ `P2/C2`: `encrypt(ID2 || N1)`
- `P2/C2` verifies encryption of `(ID1 || N2)`
  `P1/C1` verifies encryption of `(ID2 || N1)`

HOTP and TOTP

Application Domain

- Developed for user authentication
  (as part of two-factor authentication)
- http://www.openauthentication.org

HOTP

One-way authentication with SHA1-HMAC

HMAC(K,C) =
SHA1((K xor 0x5c5c...) || SHA1((K xor 0x3636...) || C))

HOTP: HMAC-based one-time password

- *HOTP* defined in IETF RFC 4226
- HOTP(K,C) = Truncate(HMAC(K,C)) & 0x7FFFFFFF
  with K a key and C a counter
- Truncate is digest-dependent 4-byte substring of a 160-bit SHA digest
- Humans who can only recall d digits use instead HOTP(K,C) mod 10^d

TOTP

One-way authentication with SHA1-HMAC

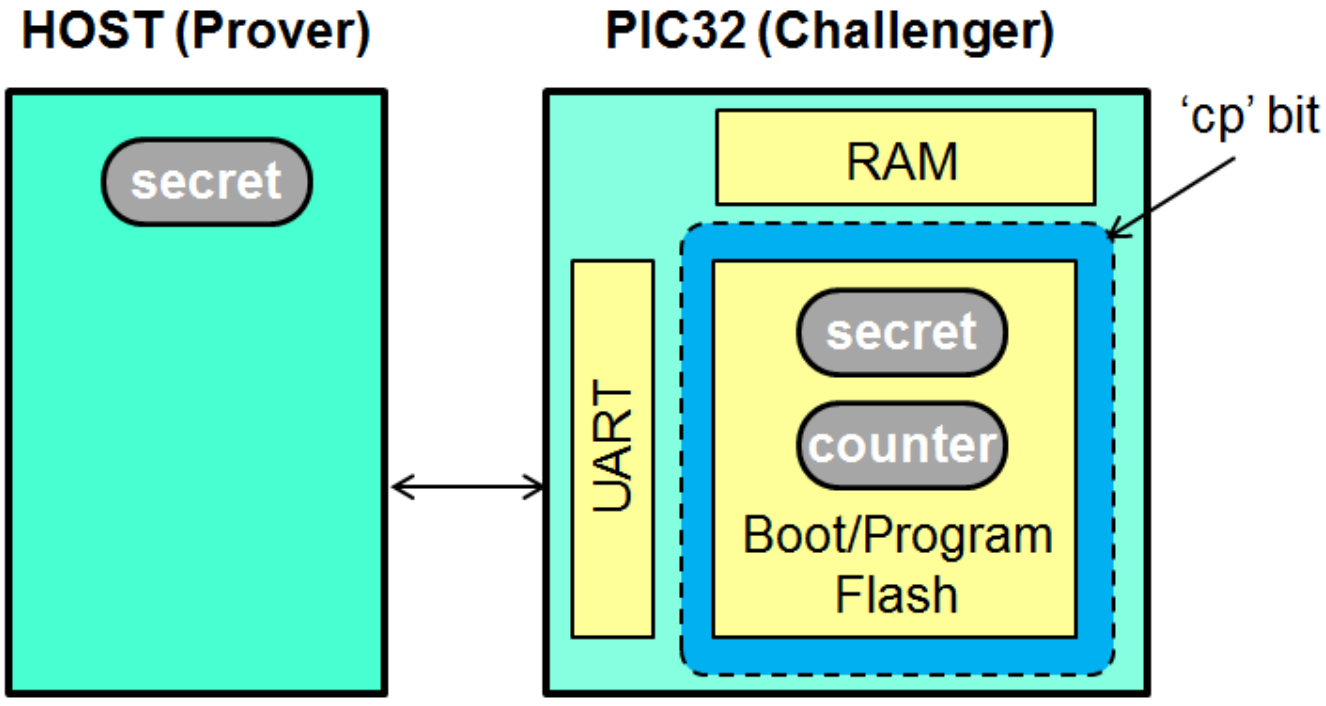
HMAC(K,C) =
SHA1((K xor 0x5c5c...) || SHA1((K xor 0x3636...) || C))

TOTP: Time based one-time password

- *TOTP*: defined in IETF RFC 6238
- TOTP(K,T) = HOTP(K,T)
  with T = floor(Unix Time / Step)
- Unix Time is the elapsed time in seconds since 00:00 UTC, 1 Jan, 1970
- Step is a time window, typically 30 seconds

1. Embedded Authentication
2. Preliminaries
   - Microcontroller Technologies
   - Basic authentication protocols
   - HOTP and TOTP
3. Authenticating Micro-controllers
   - Single-chip authentication (PIC32MX795F512L)
   - PCB-level authentication
   - Two-factor login on a watch (CC430F6137)
4. Firmware signing and verification
   - ECDSA
   - Design Flow
   - Example (ATMega2560)
5. Outlook

Single-chip scenario



Requirements

- Need persistent storage for counter
- Need protected + persistent storage for secret

Basic protocol

```
__attribute__((aligned(4096)))
const unsigned char settings[4096] =
   {0x01,0x23,0x45,0x67,0x89,0xAB,0xCD,0xEF,   // secretL
    0x10,0x32,0x54,0x76,0x98,0xBA,0xDC,0xFE,   // secretH
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};  // counter

void main() {
  ...
  hmac(settings, challenge, id, expect);

  IncCounter();

  putChallenge(challenge);
  getResponse(response);

  if (correctResponse(response, expect)) {
    // authenticated ..
  }
  ...
}
```

Writing Flash Memory

Authentication state variable stored in Flash

```
unsigned long long secret;
unsigned counter;
```

- Flash memory is persistent and (optionally) protected
- Flash memory resets to all-'1' in a block-wise operation
- Can write a '0', but not a '1' into Flash memory
- Hence, a persistent counter is tricky to implement!

Counting in Flash Memory

```
__attribute__((aligned(4096)))
const unsigned char settings[4096] =
   {0x01,0x23,0x45,0x67,0x89,0xAB,0xCD,0xEF,   // secretL
    0x10,0x32,0x54,0x76,0x98,0xBA,0xDC,0xFE,   // secretH
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};  // counter

void IncCounter() {
   int *cp, v;
   unsigned int buf[3];
   memcpy(buf, settings, 12);
   buf[2] = buf[2] + 1;
   NVMErasePage((void *) settings);
   NVMWriteWord((void *) settings, buf[0]);
   NVMWriteWord((void *) &(settings[4]), buf[1]);
   NVMWriteWord((void *) &(settings[8]), buf[2]);
}
```

Protecting Flash Memory (PIC32)

REGISTER 28-1: DEVCFG0: DEVICE CONFIGURATION WORD 0

Bit Range	Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
31:24	r-0	r-1	r-1	R/P	r-1	r-1	r-1	R/P
	—	—	—	CP	—	—	—	BWP
23:16	r-1	r-1	r-1	r-1	R/P	R/P	R/P	R/P
	—	—	—	—	PWP<7:4>			
15:8	R/P	R/P	R/P	R/P	r-1	r-1	r-1	r-1
	PWP<3:0>				—	—	—	—
7:0	r-1	r-1	r-1	r-1	R/P	r-1	R/P	R/P
	—	—	—	—	ICESEL	—	DEBUG<1:0>	

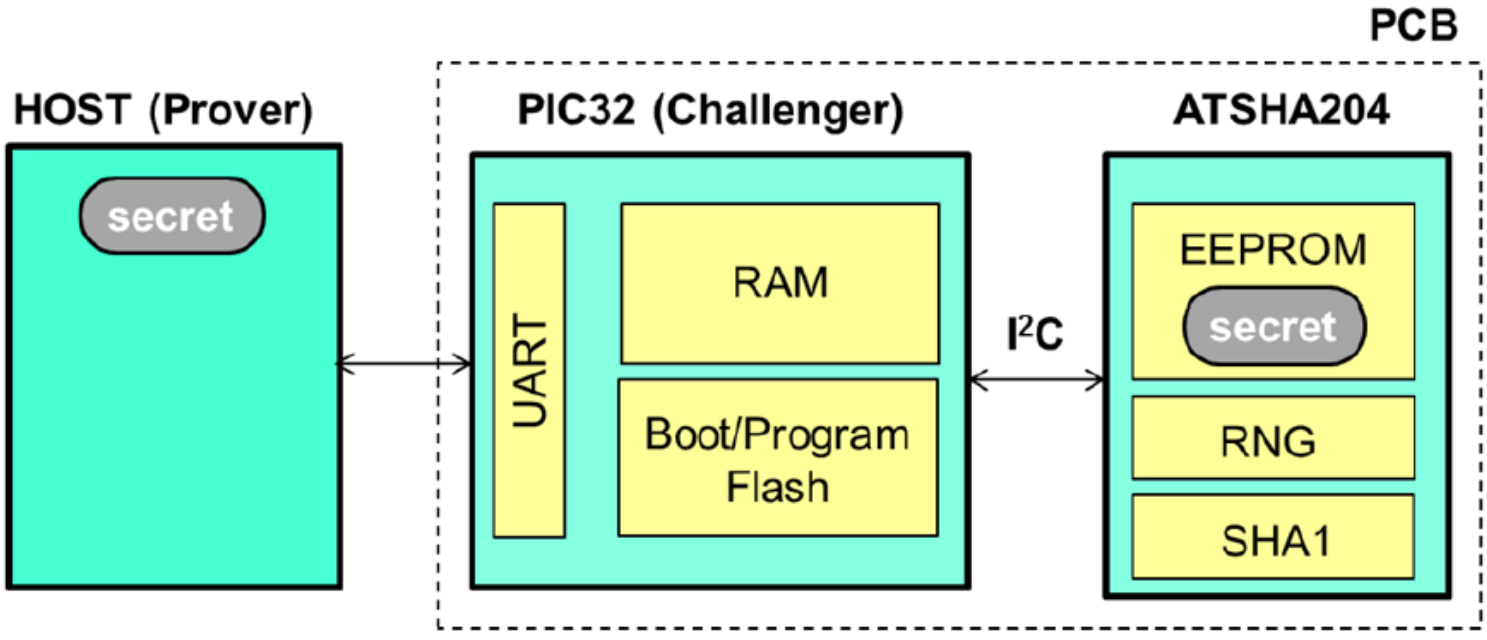
Device Configuration Registers 0 (PIC32)

- CP = Code-protect bits
- BWP = Boot-flash Write-protect bits
- PWP = Program-flash Write-protect bits

C initialization (PIC32)

```
#pragma config PWP = OFF // allow program flash write
#pragma config CP  = ON  // prevent reading of secret
```

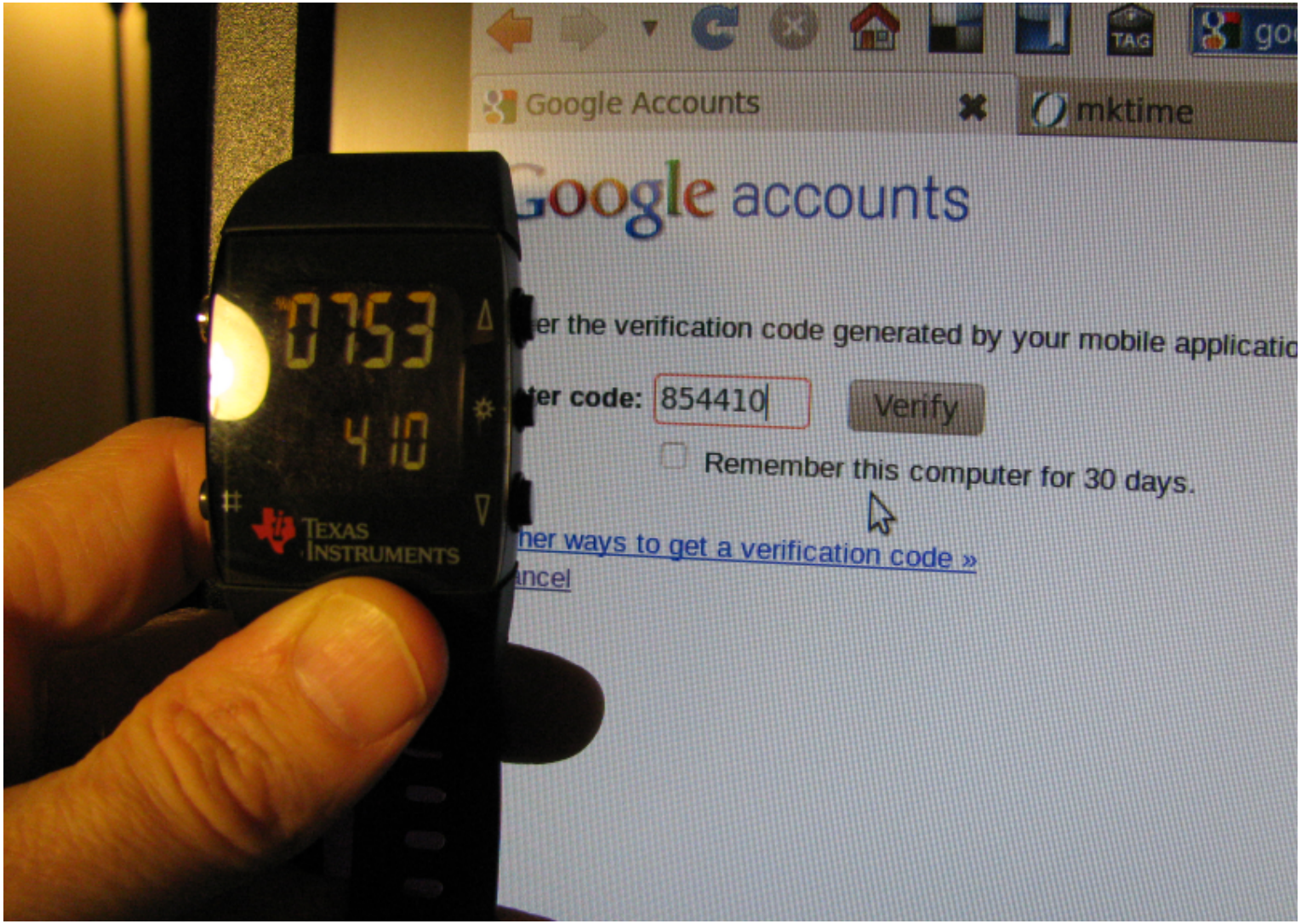
Two-chip solution



Prerequisites

- When the micro-controller non-volatile memory cannot be protected, you will need a two-chip solution.
- This solution authenticates the SHA chip (or PCB)!

Google's two-factor login



TOTP on a watch

TOTP

Recall that TOTP(K,T) = HOTP(K,T)

- The watch is always running, so can keep state in RAM
- Assuming watch is guarded, secure storage is less of an issue
- In a low-power implementation, compute TOTP only when needed (event driven, once per 30 seconds)

TOTP on a watch

```
void set_totp(u8 line) {
  // this function synchronizes the totp counter
  // to the clock time
  stotp.code = mktime(..)
                - 2208988800 // adj for unix epoch
                + 18000;     // adj for EST
  stotp.code = stotp.code / 30;
  stotp.togo = 30;  // recompute in 30 sec
  stotp.run  = 1;
}

void tick_totp() {
  // this function is called once every second
  // and adjusts the stotp time code every 30 seconds
  if (stotp.run) {
    stotp.togo = stotp.togo - 1;
    if (stotp.togo == 0) {
      stotp.code = stotp.code + 1;
      stotp.togo = 30;
    }
  }
}
```

Code signing

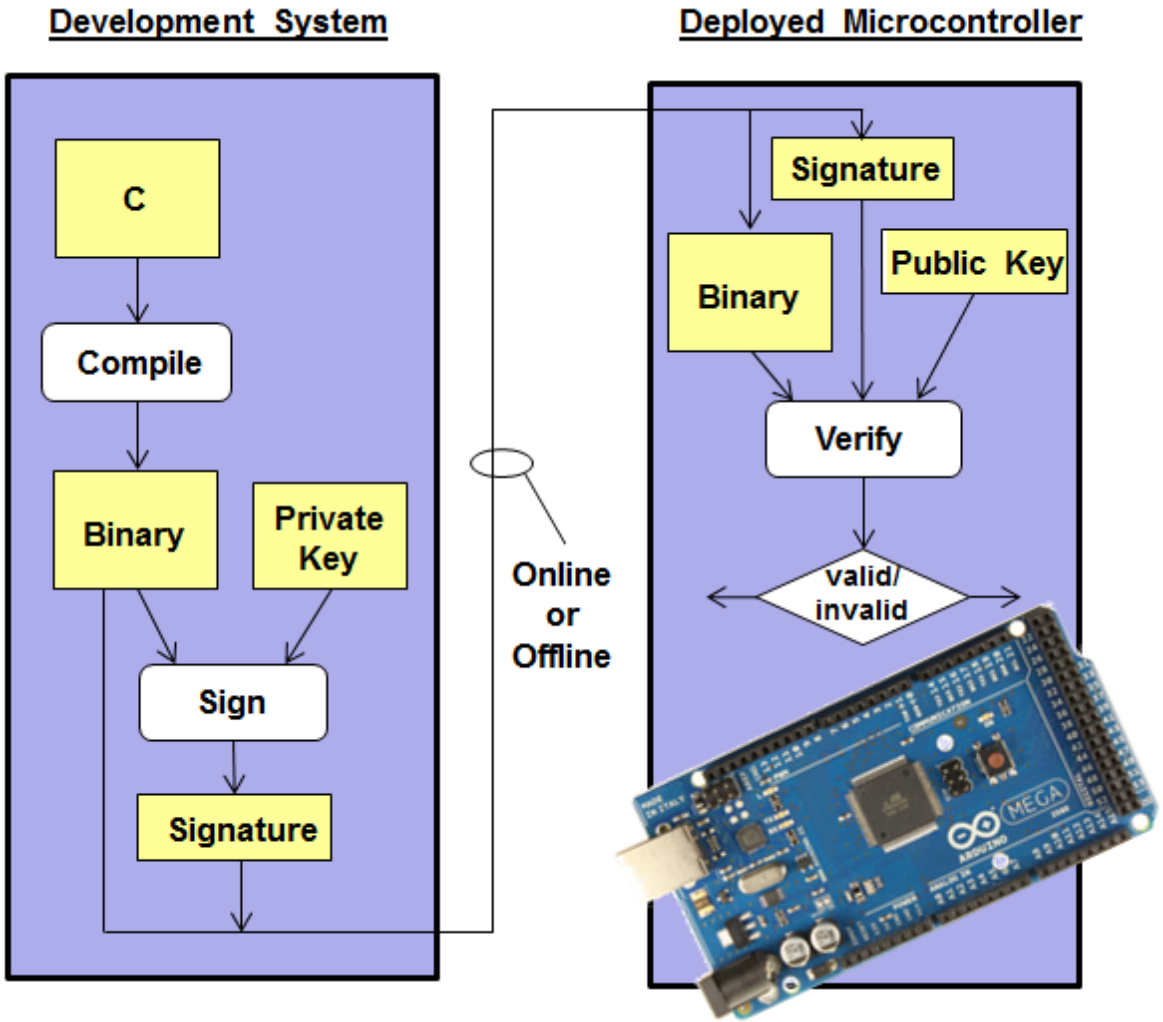
- Microcontroller authentication ensures that the hardware/firmware is genuine
- Dynamic data items or firmware plugins will need separate verification
- We will use electronic signatures (ECDSA) to verify *signed code downloads*

ECDSA

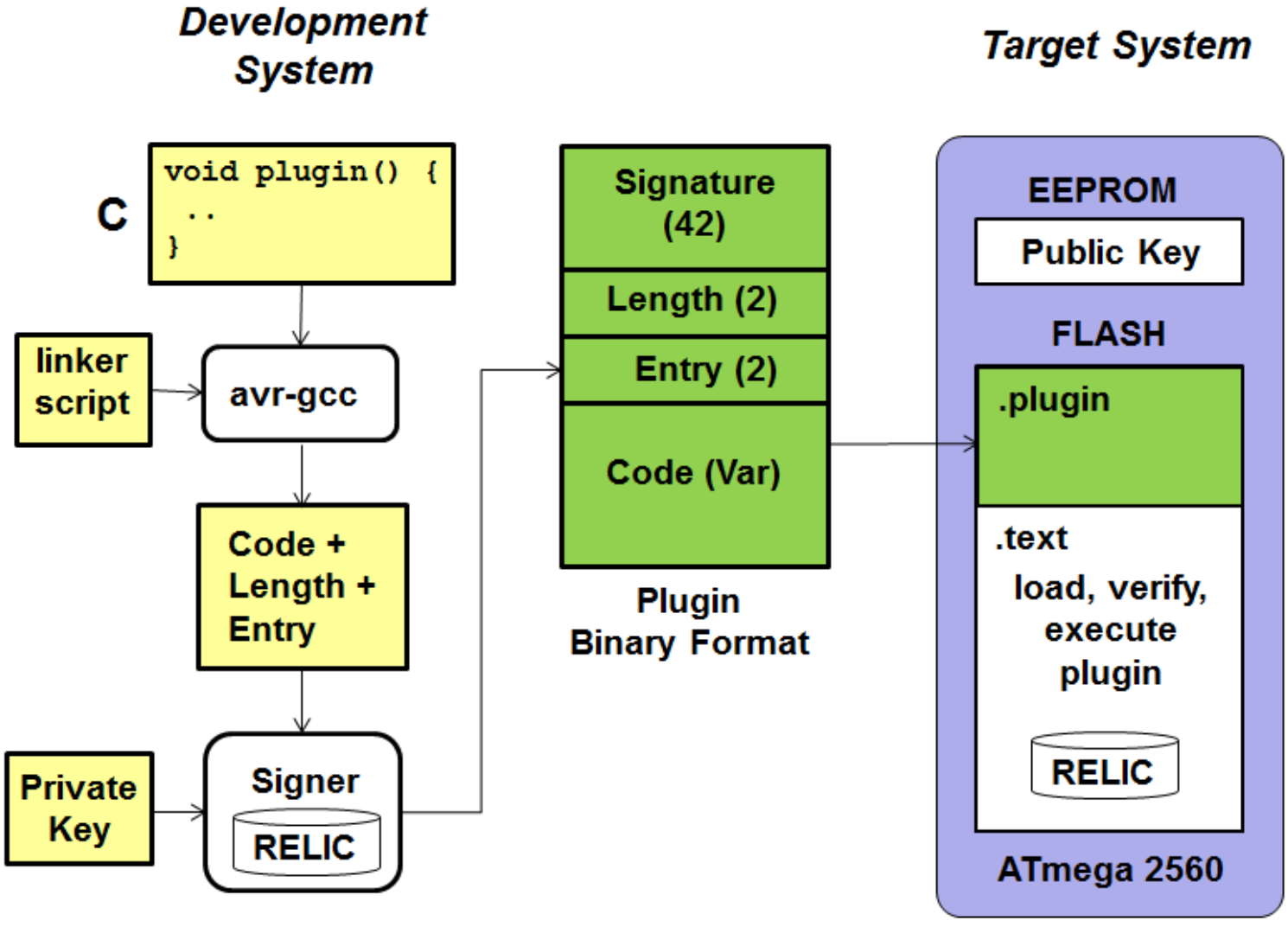
Input: Message M (or a hash of it), private key d, public key Q = d.P

Signature Generation	**Signature Verification**
Random k	$w = s^{-1} \bmod \#E$
$k.P = (x, y)$	$u_1 = M.w \bmod \#E$ $u_2 = r.w \bmod \#E$
$r = x \bmod \#E$	$u_1.P + u_2.Q = (x, y)$
$s = k^{-1}(M + d.r) \bmod \#E$	$v = x \bmod \#E$
Message: M Signature: (r, s)	Check if v = r to verify signature

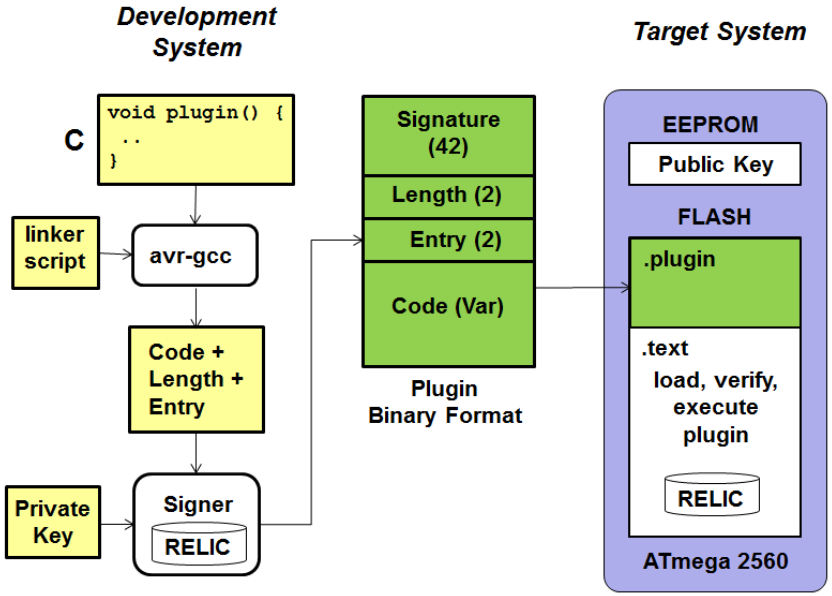
ECDSA on code plugin



Creating signed plugins



Creating signed plugins



Security Requirements

- Confidentiality for private key in development system
- Integrity for public key in target (protected flash)
- Integrity for plugin signature verification code

Plugin strategy

- Plugins will be signed with ECDSA NIST `K-163`. A signature requires 42 bytes.
- Remember that this is bare-metal programming. No OS, no runtime linking. We will therefore design the plugin as relocatable code, and use only absolute global references.
- Plugin code is inserted at *absolute target address* `0x1056`. The first 0x56 bytes will contain the signature, the length of the plugin, and the entry point.

Example plugin: a blinker

```
#include <avr/io.h>
#include <util/delay.h>

void slow(unsigned a) {
  _delay_ms(a);
}

void plugin() {
  DDRB |= _BV(DDB7);
  while (1) {
    PORTB |= _BV(PORTB7);
    slow(25);
    PORTB &= ~_BV(PORTB7);
    slow(300);
  }
}
```

Compiling the plugin

Commands

```
# compile
avr-gcc -g -v -Os -DF_CPU=16000000UL -mmcu=atmega2560 -c \
                  -o demo.o demo.c
# link (to resolve references)
avr-gcc -g -v -nostartfiles -mmcu=atmega2560 demo.o \
                  -Wl,-Map=demo.map -Wl,-T avr6.custom -o demo
# create binary image
avr-objcopy -O binary -R .eeprom demo demo.bin
```

TOTP

Uses a custom-link file, `avr6.custom`, to control location of generated code (`.text` segment) to 0x1056.

Signing the plugin

- The signer and the verifier will run on different host. The signer runs on a development system (X86), the verifier runs on an embedded system (AVR).
- We need a portable code signing/verifying library. We used RELIC (http://code.google.com/p/relic-toolkit/).
- The signer takes the binary image of the plugin as input, and generates a signed plugin in C.

Example signed plugin

```
__attribute__((__section__(".plugin")))
              unsigned char plugin[120] ={
                  0x33, 0x36, 0x42, 0x32,
                  0x44, 0x41, 0x35, 0x31, ...
};
```

Loading and Verifying the plugin

```
#include "../atmega2560_plugin_blinker/plugin.c" // plugin code

typedef void (*pluginptr_t)();
pluginptr_t verifysignature() {
  // read public key
  for (chk=0; chk<42; chk++)
    c[chk] = eeprom_read_byte(chk);
  fb_read(q->x, c, 42, 16);
  ..
  // read signature
  for (chk=0; chk<41; chk++)
    c[chk] = pgm_read_byte(&(plugin[chk]));
  bn_read_str(r, c,  41, 16);
  ..
  // verify signature
  chk = code_cp_ecdsa_ver(r, s, (PGM_P) &(plugin[86]),
                          (unsigned) len, q);
  if (chk)
    return (pluginptr_t) (&(plugin[86 + ofs]));
  else
    return 0;
}
```

Outlook

- Embedded authentication comprises (1) platform authentication and (2) authenticity of code.
- There are many implementation details that murk the waters of cryptographic protocols
  - access control and tamper resistance of key storage
  - persistent counting
  - design flows that can handle keys
- PUFs, TRNGs are just a tiny piece of the puzzle!
- For example, some interesting avenues could be
  - Embedded authentication with public-key implementation.
  - Signatures that cover variants (eg relocated code).
  - Lightweight signatures for data measurements.

Sample Projects

- P.Schaumont, "One-Time Passwords from Your Watch," Circuit Cellar 262, May 2012, `ftp://ftp.circuitcellar.com/pub/Circuit_Cellar/2012/262/Schaumont-262.zip`.
- P. Schaumont, "Electronic Signatures for Firmware Updates," Circuit Cellar 264, July 2012, `ftp://ftp.circuitcellar.com/pub/Circuit_Cellar/2012/264/Schaumont-264.zip`.
- P. Schaumont, "Embedded Authentication," Circuit Cellar 270, February 2013, `ftp://ftp.circuitcellar.com/pub/Circuit_Cellar/2013/270/270-Schaumont.zip`.