캡스톤 디자인, 2016/11/01

임베디드 오픈 플랫폼 기반의 네트워크를 통한 펌웨어 보안성 검증 도구

상명대학교 컴퓨터소프트웨어공학과

201321301 문지연

201321312 위사랑

목 차

개요

- 배경
  - 사물인터넷 시대가 도래하면서 운영체제나 응용 레벨에서 탐지가 불가능한 펌웨어 루트킷/ 변조의 위험성 증가

→ 동작중인 펌웨어 보안성 검증 필요

개요

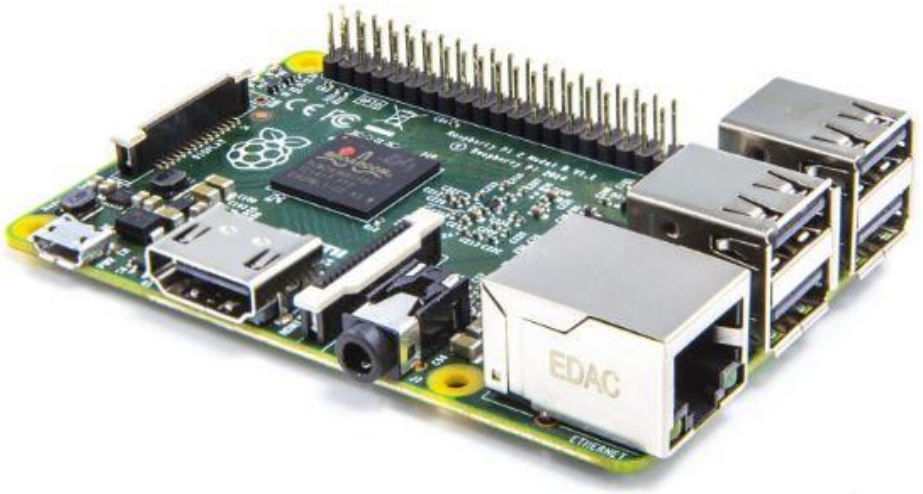
- 배경
  - 펌웨어 해킹 사례
    - 공유기
      - 2012년, IPTIME 유·무선 공유기 펌웨어 해킹 사례 발표[1]
      - 공격자가 사용자의 허가 없이 네트워크 조작 및 악성코드 유포
    - 차량
      - 2015 데프콘에서 차량 네트워크 해킹 사례 발표[2]
      - 지프차인 '체로키(Cherokee)'를 해킹하여 브레이크, 시동 등 차량을 외부에서 조작
    - USB
      - 2014 블랙햇에서 "BadUSB" 라는 이름으로 해킹사례 발표[3]
      - 공격자가 특정 USB장치를 조작하여 해킹용 도구로 만들어 공격

[1] http://returnaddr.org/b0d/view.php?id=mydoc_secudoc&no=6
[2] Charlie Miller, and Chris Valasek "Remote exploitation of an unaltered passenger vehicle," *DEFCON,* Aug 2015.
[3] Karsten Nohl and Jakob Lell, "BadUSB - on accessories that turn evil," Black Hat USA, Aug. 2014

개요

- 배경
  - 라즈베리파이2란

    “컴퓨터 교육 및 취미 활동 증진을 위해 만들어진 ARM 리눅스 기반싱글 보드 컴퓨터”



개요

- 목표
  - 펌웨어의 보안성을 검증할 수 있는 도구 개발
  - 기존의 검증도구 보완

기존의 검증도구와의 비교

- 기존의 라즈베리파이2의 무결성 검증도구와의 비교

	기존의 검증도구	제안하는 검증도구
검증 시점	운영체제 부팅전	운영체제 부팅전
검증도구에 사용하는 암호 알고리즘	**CRC**	**SHA1, ECDSA**
검증 대상	플래시메모리 내의 펌웨어 파일	플래시메모리 내의 펌웨어 파일
검증 도구 구현	U-boot 부트 프로시저	U-boot 부트 프로시저
검증 도구 위치	플래시 메모리 내	플래시 메모리 내
순정 파일의 위치	**플래시 메모리 내**	**벤더사가 소유**
순정파일의 무결성	**확인 불가능**	**서명을 통해 확인가능**

기존의 검증도구와의 비교

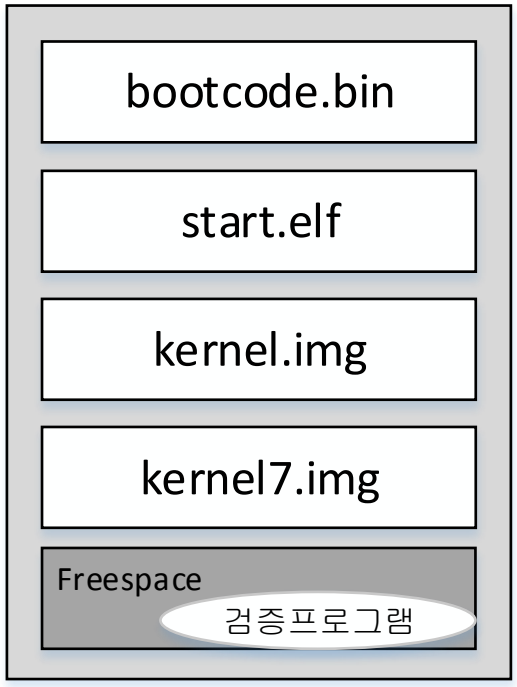
- 보완 사항
  - 무결성을 검증하기 위해 비교하는 순정 CRC가 임베디드 플랫폼 내부에 존재하기 때문에, 순정 CRC의 변조 위험 가능성이 존재
    - 네트워크를 통한 검증(TFTP)
  - 기존의 검증 도구는 검증 도구자체에 대한 순정 확인 불가
    - 공개키를 이용한 서명(ECDSA)
  - CRC는 역산이 가능하므로 안전하지 않음
    - 일방향 해시함수 사용(SHA1)

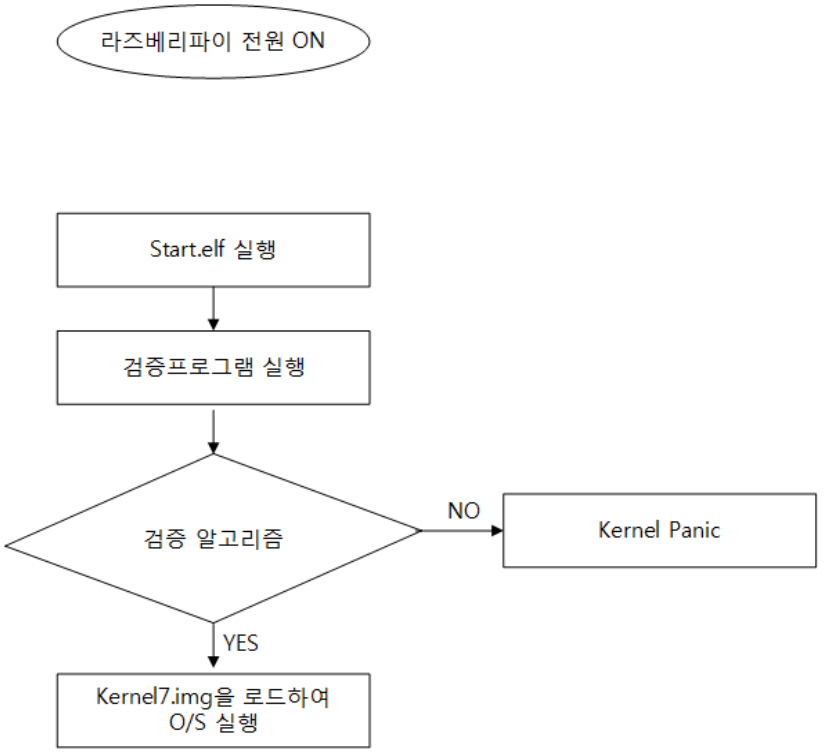
네트워크를 통한 펌웨어 무결성 검증

- 검증 시점: 운영체제 부팅 전
- 검증 도구에 사용하는 기법: SHA1, ECDSA
- 검증 대상: 플래시 메모리 내의 펌웨어 파일
  - bootcode.bin : 첫번째 부트로더
  - start.elf : 두번째 부트로더
  - kernel7.img : 리눅스 커널
  - kernel.img : 보조 커널
- 검증 도구 구현: U-boot 부트 프로시저
- 검증 도구의 위치: 플래시 메모리 내

네트워크를 통한 펌웨어 무결성 검증

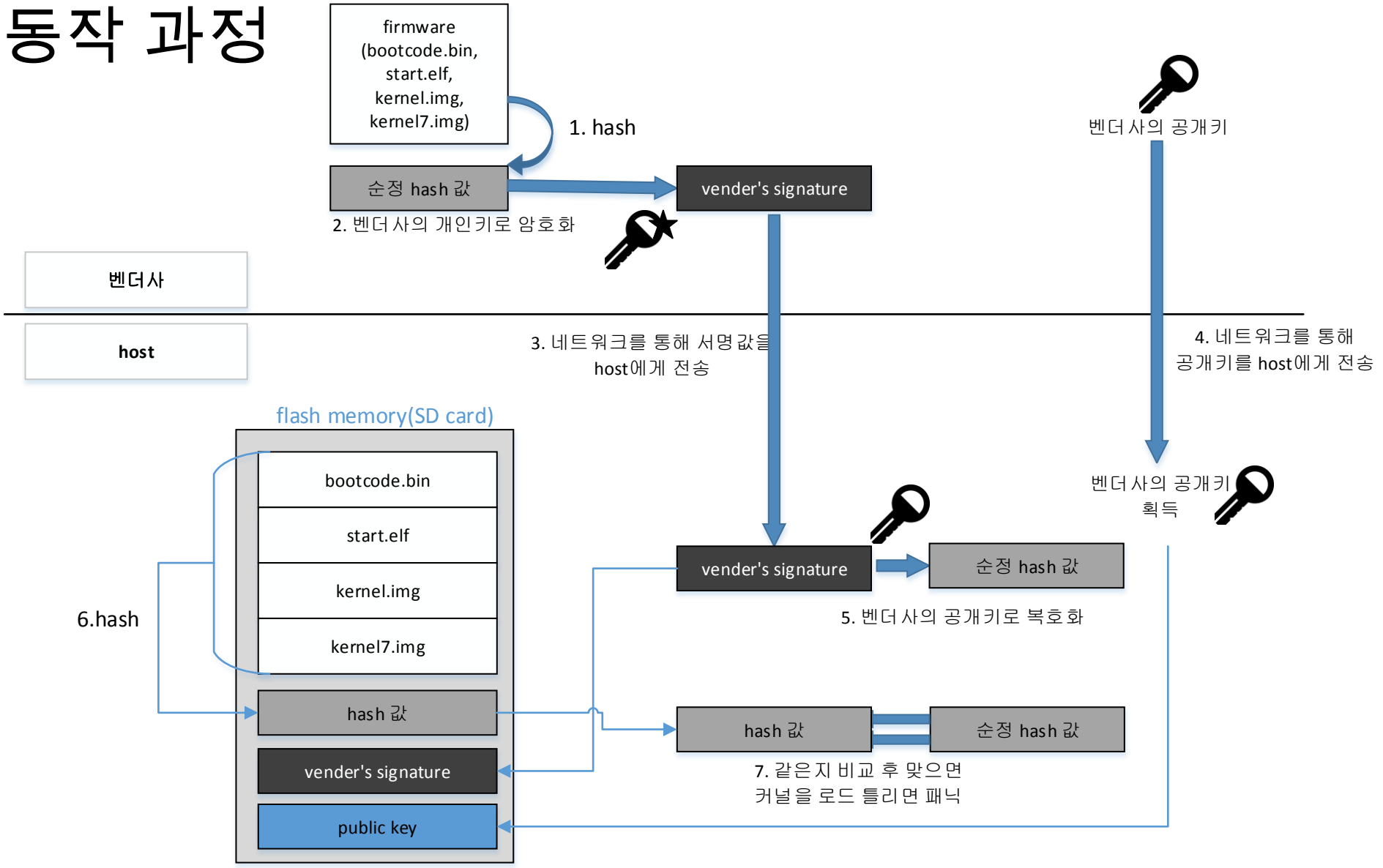
- U-Boot를 통한 부트 프로시저
  - 3rd bootloader(start.elf)가 커널(kernel7.img)을 로드하기 전에 4th bootloader(u-boot.bin)가 동작





네트워크를 통한 펌웨어 무결성 검증

- 동작 과정



네트워크를 통한 펌웨어 무결성 검증

- 동작 과정
  - 벤더사는 SHA1을 통해 펌웨어의 hash 값을 생성한 후, ECDSA(Elliptic Curve Digital Signature Algorithm)를 통해 자신의 개인키로 hash 값을 서명
  - 벤더사는 네트워크를 통해 서명값과 공개키를 검증 플랫폼에게 전달
  - 검증 플랫폼은 수신한 공개키로 서명값을 검증한 후, 펌웨어의 hash 값을 생성 및 비교
  - 비교결과
    - 같을 경우 :커널(kernel7.img) 실행
    - 다를 경우 : 메시지 출력 후 패닉(시스템 구동 중지)

개발진행사항

- 4월 진행 사항(자료조사)

1. 부트되기 전, U-boot에서 네트워크 사용이 가능한지 찾아 볼 것
2. 네트워크를 통해 받은 인증서를 통해 공개키를 추출하여 사용할 수 있는 코드를 찾아볼 것
3. 공개키를 이용해 제조사가 보낸 순정 값에 붙은 서명을 확인하는 과정이 담긴 코드를 찾아볼 것

개발진행사항

- 6월 진행 사항(U-boot 환경설정 및 개발시작)

  1. U-boot Command line Interface 설정
  2. 네트워크를 통한 파일 전송 (TFTP)

개발진행사항

- 6월 진행 사항(U-boot 환경설정 및 개발시작)

1. U-boot Command Line Interface 설정
   - U-boot Command Line 환경 설정 시,
     USB KEYBORAD 동작 하지 않음

CODE: SELECT ALL

```
USB low-/full-speed (USB 1.0) devices do not currently work. USB Ethernet devices are
typically high-speed and hence work, and USB keyboards are typically low-/full-speed devices
and hence do not currently work
```

It also states

CODE: SELECT ALL

```
To use U-Boot and Linux as a development system and to make full use of all their
capabilities you will need access to a serial console port on your target system. Later, U-
Boot and Linux can be configured to allow for automatic execution without any user
interaction.

There are several ways to access the serial console port on your target system, such as using
a terminal server, but the most common way is to attach it to a serial port on your host.
Additionally, you will need a terminal emulation program on your host system, such as cu or
kermit
```

It seems that direct keyboard access might not be possible

개발진행사항

- 6월 진행 사항(U-boot 환경설정 및 개발시작)

1. U-boot Command Line Interface 설정
   - Raspberry Pi 2 원격 접속 시도
   - PL2303 3.3v 전용 USB-to Serial 장치를 통한 원격 접속
   - http://blog.xcoda.net/83

개발진행사항

- 6월 진행 사항(U-boot 환경설정 및 개발시작)

2. 네트워크를 통한 파일 전송 (TFTP – Raspberry Pi 2)
   - Network 설정

```
U-Boot> setenv ipaddr 192.168.0.27
U-Boot> setenv gatewayip 192.168.0.1
U-Boot> setenv netmask 255.255.255.0
U-Boot> saveenv
```

```
U-Boot> setenv serverip 192.168.0.20
U-Boot> saveenv
```

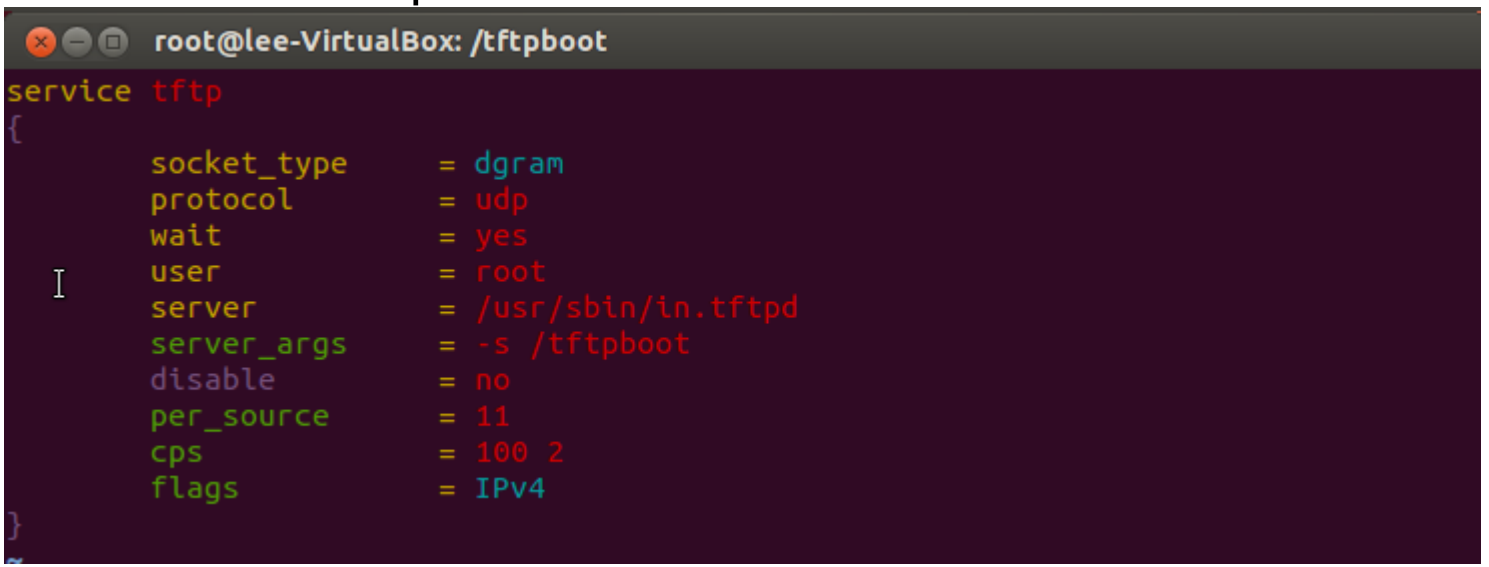
   - IP Test

```
U-Boot> ping 192.168.0.20
Waiting for Ethernet connection... done.
Using sms0 device
host 192.168.0.20 is alive
```

개발진행사항

- 6월 진행 사항(U-boot 환경설정 및 개발시작)

2. 네트워크를 통한 파일 전송 (TFTP – Host PC)
   - TFTP 설치
     - apt-get install tftp tftpd xinetd
   - TFTP 설정 파일 수정
     - vi /etc/xinetd.d/tftp



개발진행사항

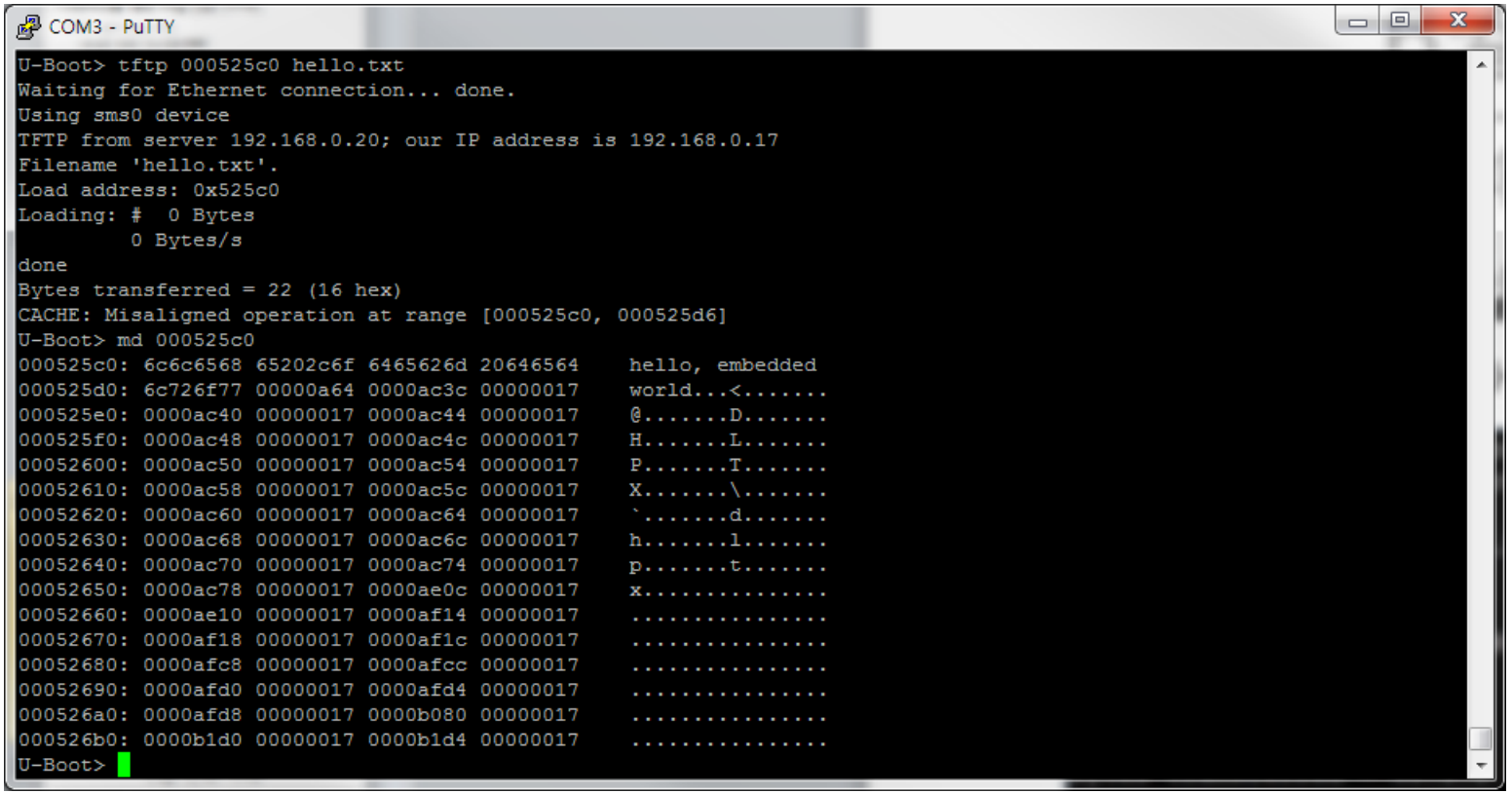
- 6월 진행 사항(U-boot 환경설정 및 개발시작)

  2. 네트워크를 통한 파일 전송 (TFTP)
     - TFTP 디렉토리 생성 및 권한 설정
       - mkdir /tftpboot
       - chmod –r 777 /tftpboot
       - service xinetd restart

개발진행사항

- 6월 진행 사항(U-boot 환경설정 및 개발시작)

2. 네트워크를 통한 파일 전송 (TFTP)
   - TFTP를 이용한 파일 전송 테스트



개발진행사항

- 6월 진행 사항(U-boot 환경설정 및 개발시작)

2. 네트워크를 통한 파일 전송 (TFTP)

- void main_loop(void)

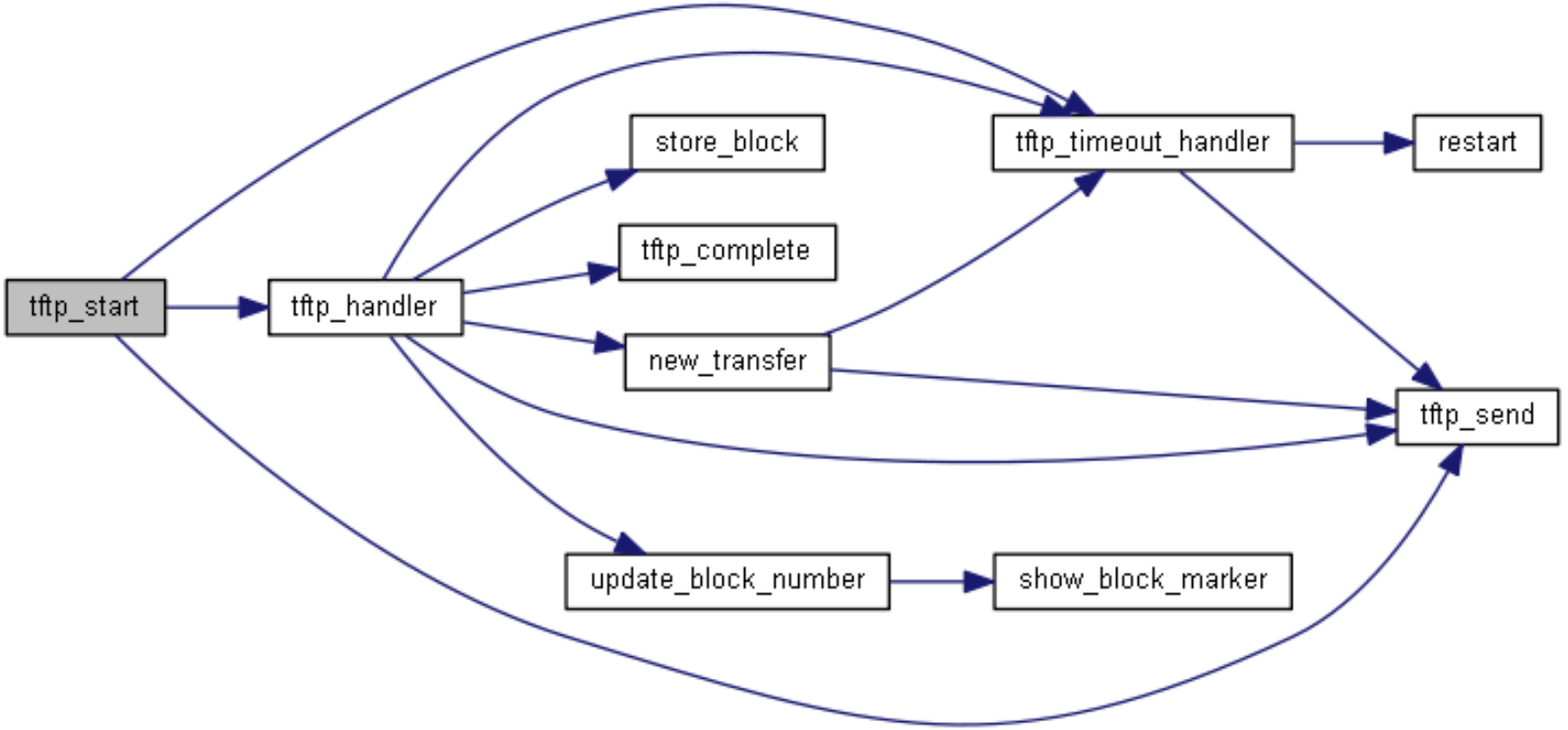
```
#include <command.h>
#include <net.h>

void main_loop(void) {
  // tftp
  printf("\n# Receive the file using TFTP\n");
  cmd_tbl_t *bcmd;
  bcmd = find_cmd("tftpboot");
  char *tftpboot[5] = {"tftpboot", "0x00200000", "hash.txt"};
  do_tftpb(bcmd, 1, 3, tftpboot);

  char *ptr = (char *)0x200000;
  printf("Value: %s \n", ptr);
}
```

개발진행사항

- 6월 진행 사항(U-boot 환경설정 및 개발시작)

2. 네트워크를 통한 파일 전송 (TFTP)

- tftp.c 동작 과정(u-boot/net/tftp.c)



개발진행사항

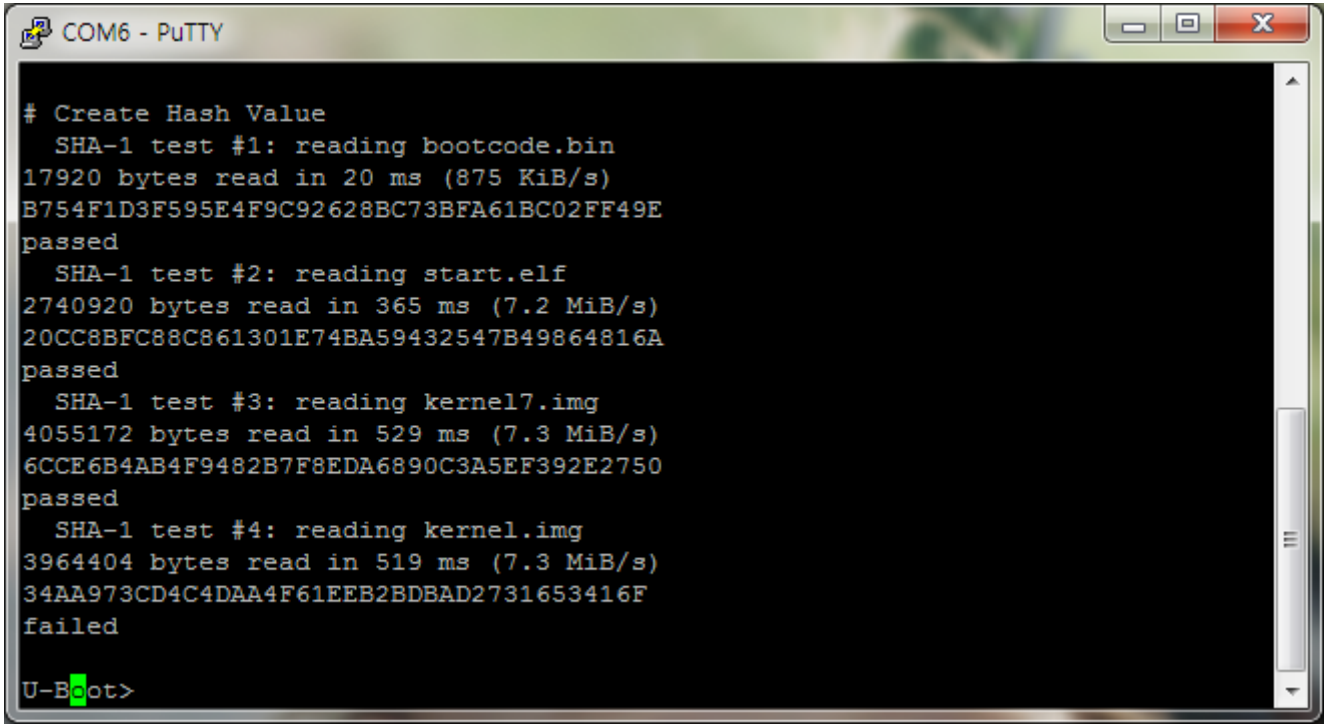
- 9월 진행 사항

1. SHA1을 이용한 펌웨어 hash 값 생성
2. 벤더사는 자신의 개인키와 공개키 쌍을 생성한 후, 펌웨어 파일에 대한 hash 값 서명

개발진행사항

- 9월 진행 사항

1. 펌웨어 hash 값 생성
   - SHA1을 통한 hash 값 생성 테스트



개발진행사항

- 9월 진행 사항

1. 펌웨어 hash 값 생성

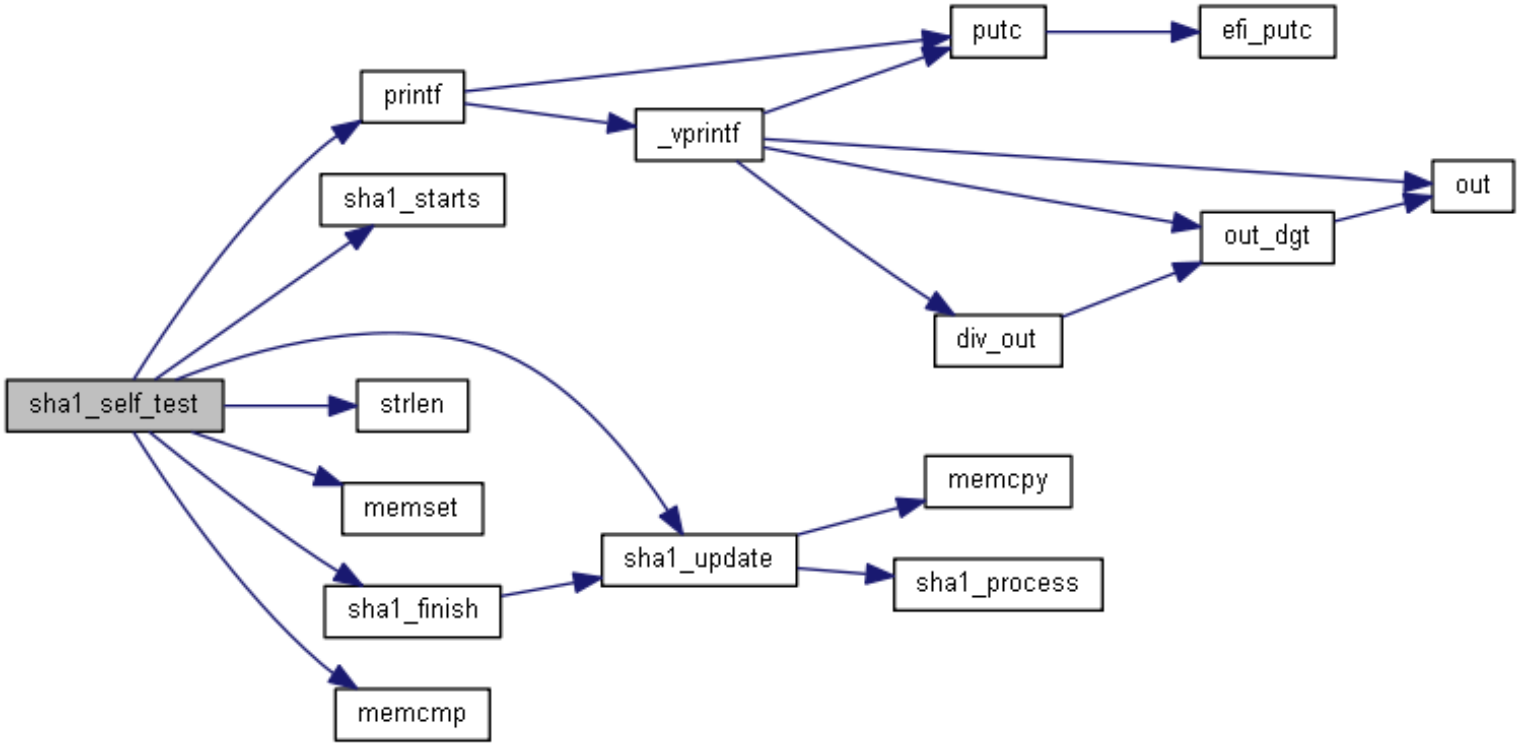
```
#include <fs.h>
#define load_file1   "bootcode.bin"
#define load_file2   "start.elf"
#define load_file3   "kernel7.img"
#define load_file4   "kernel.img"
#define load_addr   "0x00200000" // SDRAM의 물리적 주소

int sha1_self_test (void) {
   int firm_size = 0; // init file size
   cmd_tbl_t *bcmd; // struct for using command
   char *target_load_addr = (char *)0x00200000;
   bcmd = find_cmd("fatload");

   for (i = 0; i < 4; i++) {
      printf ("  SHA-1 test #%d: ", i + 1);
      sha1_starts (&ctx);
      char *loadimg[5] = {"fatload", "mmc", "0", load_addr};
```

```
if (i == 0) {
   loadimg[4] = load_file1;
   // origin_hash = firm_orig_hash1;
} else if (i == 1) {
   loadimg[4] = load_file2;
   // origin_hash = firm_orig_hash2;
} else if (i == 2) {
   loadimg[4] = load_file3;
   // origin_hash = firm_orig_hash3;
} else {
   loadimg[4] = load_file4;
   // origin_hash = firm_orig_hash4;
}

firm_size = do_load(bcmd, 0, 5, loadimg, FS_TYPE_FAT);
if (i != 3)
   sha1_update (&ctx, (unsigned char *)target_load_addr, firm_size);
else {
   memset (buf, 'a', 1000);
   for (j = 0; j < 1000; j++)
      sha1_update (&ctx, buf, 1000);
}

sha1_finish (&ctx, sha1sum);
if (memcmp (sha1sum, sha1_test_sum[i], 20) != 0)
}
```

개발진행사항

- 9월 진행 사항

1. 펌웨어 hash 값 생성
   - sha1.c 동작 과정(u-boot/lib/sha1.c)



개발진행사항

- 9월 진행 사항

2. RSA를 이용한 서명

- 보안 강도에 따른 RSA 암호 알고리즘[4]

보안강도	인수분해문제 (비트) – RSA
80비트	1024
112비트	2048
128비트	3072
192비트	7680
256비트	15360

- 현재 u-boot내에 존재하는 자료형은 int가 32bit로 제일 큼
- 그러나 위의 표를 보면 RSA의 키의 길이가 최소 512byte이상이 되어야 함
- RSA의 512byte의 키 길이를 연산하여 사용할 수 있는 자료형이 없음
  - 즉, u-boot의 자료형을 사용하게 되면 오버플로우 발생
- 큰 숫자를 이용할 수 있는 자료형을 구현해야 함
- 소인수분해를 해야하기 때문에 속도 최적화 필요

[4] 암호 알고리즘 및 키 길이 이용 안내서, 한국인터넷진흥원, 2013

개발진행사항

- 9월 진행 사항

2. RSA를 이용한 서명
   - 문제를 해결하기 위한 두 가지 대안
     1. 큰 수를 계산하기 위한 라이브러리 찾기
        - GMP(GNU Multiple-Precision)[5]
        - LibTomCrypt[6]
        - FLINT/C[7]

     2. 서명 알고리즘 바꾸기

[5] https://gmplib.org/
[6] http://www.libtom.org/LibTomCrypt/
[7] Michael Welschenbash, "Crptography in C and C++ Second Edition, " Apress, 2001

개발진행사항

- 9월 진행 사항

2. ECDSA를 이용한 서명
   - ECDSA
     - Elliptic Curve Digital Signature Algorithm의 약자
     - RSA암호에 비해 적은 키 길이를 요구
     - 수행속도 빠름
   - RSA vs ECDSA 키 길이 비교[4]

보안강도	RSA(소인수분해)	ECDSA(이산대수)
80비트	1024	160
112비트	2048	224
128비트	3072	256
192비트	7680	384

개발진행사항

- 9월 진행 사항

  2. ECDSA를 이용한 서명
     - 서명 값 생성 및 검증 테스트

```
uint8_t private[NUM_ECC_DIGITS] = {A5 79 67 C1 43 75 03 DF 97 E4 9A 3C F6 1E CB CA 9E 64 D4 72 E1 F
B B5 79 };
EccPoint public =
    {61 33 08 0D BB AE 5D C9 A7 AB D2 AB 8F 64 39 90 6C 9A 7C AF B4 58 C8 D2 },
    {35 FB 95 3B 5F F1 47 DF 92 32 8E 20 7D DA 30 7F 70 67 0C 48 5F 12 E7 0F }};

------------------------------------------------------------
hash = B7 54 F1 D3 F5 95 E4 F9 C9 26 28 BC 73 BF A6 1B C0 2F F4 9E
-----------
bootcode.bin's signed value
r = F1 81 68 AC 1C 5B 66 39 9E E1 22 9A 15 3F 97 49 F1 B3 03 3E 5F F3 4A AD
s = 5C 1B 50 55 E6 B2 91 38 BE 0C C1 70 E2 3F 91 B9 F1 5E 75 1E E2 06 5E 95
---------------verify---------------
recived data = F1 81 68 AC 1C 5B 66 39 9E E1 22 9A 15 3F 97 49 F1 B3 03 3E 5F F3 4A AD
      v      = F1 81 68 AC 1C 5B 66 39 9E E1 22 9A 15 3F 97 49 F1 B3 03 3E 5F F3 4A AD
verify OK
```

개발진행사항

- 차후 진행사항
  - 개별 구현된 함수를 병합
  - 동작환경 테스트

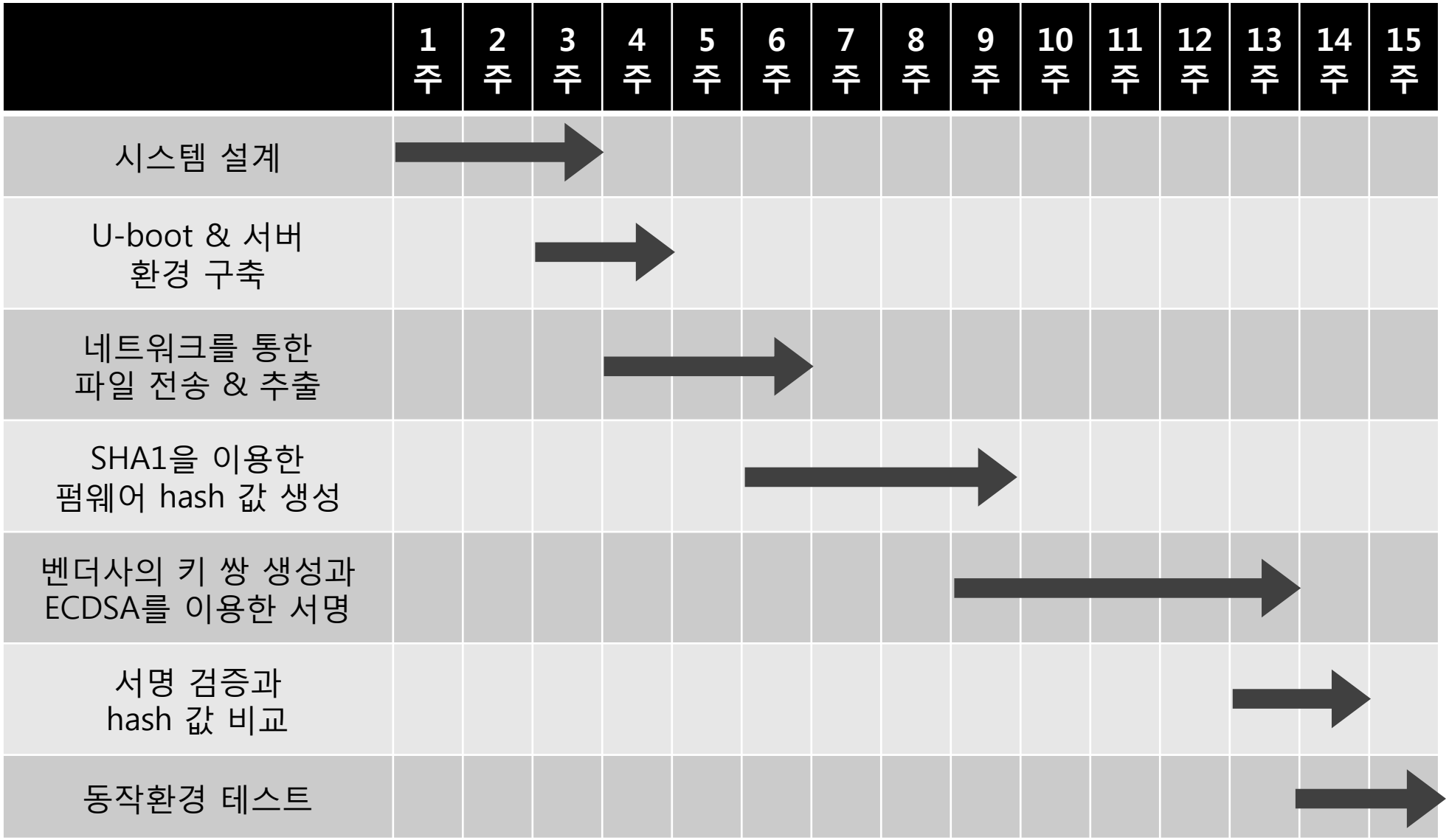
시연 영상

기대효과 및 활용방안

- 펌웨어의 보안성 강화
  - 펌웨어는 현재 대부분의 전자기기에서 사용
    (ex. PC, 스마트폰, 공유기, 프린터, 라우터, USB 등)
  - 기기 스스로 펌웨어 무결성 검증 가능

- 다른 임베디드 오픈 플랫폼에 적용 가능

개발 일정 및 개발 환경


1 주
2 주
3 주
4 주
5 주
6 주
7 주
8 주
9 주
10 주
11 주
12 주
13 주
14 주
15 주
시스템 설계
U-boot & 서버 환경 구축
네트워크를 통한 파일 전송 & 추출
SHA1을 이용한 펌웨어 hash 값 생성
벤더사의 키 쌍 생성과 ECDSA를 이용한 서명
서명 검증과 hash 값 비교
동작환경 테스트

개발 환경

- PC(Host)
  - OS : Ubuntu 16.04
  - CPU : x86
  - U-boot : 2016.07
- 라즈베리파이 2(Target)
  - OS : Raspbian jessie
  - CPU : ARM

Q & A